Implementación de un gestor de bases de datos orientado a objetos

UNIVERSIDAD
COMPLUTENSE
MADRID

Víctor Delgado Fernandez
Carlos Fernández Bravo
Álvaro Isabel Torija

Director: Manuel Montenegro Montes
2016/2017

Trabajo de fin de grado del Grado en Ingeniería Informática

Índice

Resumen y palabras clave

Resumen

El objetivo principal de este trabajo es la presentación de una librería que permita al programador de una aplicación abstraerse del mecanismo de almacenamiento de objetos en una base de datos. Hay librerías similares que han dejado de existir por diferentes razones y librerías existentes que a nuestro parecer son más complejas de utilizar en un entorno educativo, que es la finalidad para la que se ha propuesto nuestro proyecto.

Nuestro proyecto consiste, por tanto, en el desarrollo de una librería en Java para manejar objetos en la base de datos, los cuales se podrán insertar, actualizar, borrar y consultar sin necesidad de conocer el lenguaje SQL ni sus detalles de almacenamiento en la base de datos. Para ello tan solo es necesario incluir nuestra librería como dependencia en un proyecto Java, y disponer de un SGBD relacional.

El resultado del proyecto ha cumplido todas nuestras expectativas. La librería soporta relaciones entre objetos con distintos tipos de multiplicidad, que formaban parte del objetivo inicial del proyecto.

Palabras clave

Librería, bases de datos, modelo relacional, reflexión, SQL, Java, programación orientada a objetos, relaciones de multiplicidad entre objetos.

Abstract & Keywords

Abstract

The main objective of this project is the creation of a library that allows the programmer of an application to abstract himself from the object storage mechanisms in a database.
There are already libraries that fulfill this purpose, but for different reasons they are not maintained anymore. There are also already existing libraries that in our opinion are very complex to use in an educative environment, which is the goal of our project.

Therefore, our project consists in the development of a Java library to manage objects in a database. Objects can be inserted, updated, deleted and retrieved without the need of knowing neither the SQL language nor the storage details in a database. Therefore, all that you need is to include our library as a Java project dependency and have a relational DBMS.

The outcome of the project has accomplished all of our expectations. The library support relations between objects with different kind of multiplicity, which was part of the initial project objective.

Key words

Library, database, relational model, reflection, SQL, Java, object-oriented programming, multiplicity relations between objects.

Capítulo 1: Introducción

1. Antecedentes

Los lenguajes orientados a objetos aún tienen una cuota de mercado significativa, al igual que las bases de datos relacionales. Las aplicaciones utilizan cada vez más sistemas gestores de bases de datos para almacenar información. Tradicionalmente se utilizaba SQL integrado directamente en el código de la capa de acceso a base de datos de la aplicación, pero este enfoque es poco mantenible, por ser de demasiado bajo nivel y mantener un fuerte acoplamiento entre el diseño de la BD y la aplicación. Para solventar este problema existen mecanismos de abstracción que permiten unir el modelo de objetos, que es con el que trabaja el programador, con el modelo relacional, que es aquel con el que trabaja la base de datos. Este proyecto aborda la integración entre la capa de acceso a base de datos y el resto de aplicaciones. Nuestro objetivo es diseñar una librería que abstraiga al programador de los estilos de almacenamiento de la base de datos.

Nuestra librería se propone como alternativa a varias soluciones ya existentes que exponemos a continuación:

- **Hibernate**: Es una librería para poder conectarse a un gestor de base de datos relacional. Permite al desarrollador, mediante anotaciones en el propio código fuente, definir la correspondencia entre su modelo de datos orientado a objetos y las tablas de la base de datos en las que se guardarán dichos objetos. Además integra un lenguaje parecido a SQL, llamado HQL, con el que se pueden hacer consultas en la base de datos recuperando los objetos guardados.[1]
- **db4o**: Es un gestor de base de datos nativo orientado a objetos que no se basa en ejecutar consultas, sino en el uso de los métodos de una librería para guardar, recuperar y borrar objetos. No se mantiene desde 2008.[2]
- **ORMLite**: Es una librería muy parecida a Hibernate que también requiere anotaciones en el código fuente, pero es más sencilla y mucho más limitada. Está orientada principalmente al desarrollo en plataformas Android.[3]
- **OJB**: Es una librería para mapear objetos en una base de datos relacional usando un documento XML. Esta librería ya no se mantiene.[4]
- **ObjectStore**: Es una librería para C++ que proporciona una base de datos nativa orientada a objetos. Para la construcción de objetos es necesario modificar el constructor para que este objeto se guarde a la vez en la base de datos y en memoria. Se detectan los cambios en el objeto para actualizar la base de datos.[5]

Todas las tecnologías aquí enumeradas resuelven en mayor o menor medida el problema de abstraer los detalles internos de una base de datos. Sin embargo, nosotros queremos que nuestra librería funcione en Java y que el programador no se vea obligado a introducir anotaciones en el código para asociar sus objetos con las tablas de la base de datos. La

librería db4o cumple con estos requisitos pero desgraciadamente ya no se mantiene. El objetivo del proyecto es basarnos en estas ideas para desarrollar una librería que tenga una interfaz similar para el programador pero que utilice una base de datos relacional como mecanismo de almacenamiento.

2. Objetivos

2.1. Diseño de API para librería BD orientada a objetos

El primer y principal objetivo de este proyecto es diseñar e implementar una API para una librería de bases de datos orientada a objetos.

Como mecanismo de almacenamiento subyacente a nuestra librería se hará uso de una base de datos de tipo relacional, aunque el modelo de datos de almacenamiento es totalmente transparente al programador, y podría haberse utilizado otro distinto (por ejemplo, una base de datos XML o MongoDB).

Mediante esta API pretendemos que el usuario final de la misma pueda realizar todas las gestiones relacionadas con bases de datos tales como inserción, actualización, borrado o carga sin necesidad de construir él mismo las sentencias SQL necesarias para dichas gestiones.

Para realizar las operaciones de inserción, actualización y borrado el usuario dispondrá de unos métodos de la librería que se encargarán de cada uno de los procesos. Estos métodos reciben, directamente como parámetro, el objeto a insertar, actualizar o eliminar, sin necesidad de indicar información sobre tablas o columnas de la base de datos sobre las que realizar la operación. Para las operaciones de carga de objetos de la base de datos se diseñará un sistema de consultas con restricciones que se apoyará en la librería para su ejecución. Los mecanismos de consulta propuestos devuelven objetos ya construidos a partir de las filas correspondientes de la BD.

Como primer objetivo, los objetos con los que se podrá operar en un principio solo constan de atributos de tipo básico (enteros, cadenas, etc.). Se proporcionarán los mecanismos necesarios para gestionar la identidad de los objetos, de manera que, por ejemplo, si se guarda un mismo objeto varias veces, sólo se almacene una representación en la base de datos.

2.2. Extensión para soportar relaciones entre objetos

Para mejorar las operaciones de inserción, actualización, borrado y recuperación de objetos nombradas en el objetivo anterior se extenderá la API para soportar objetos que tengan referencias a otros objetos de cualquier tipo no básico, excepto objetos de tipo contenedor como listas, conjuntos o arrays.

Para implementar de forma correcta este objetivo se razonará la forma de construir las tablas SQL para representar las relaciones entre objetos en la base de datos relacional, y se proporcionarán mecanismos para recuperar cadenas de referencias de objetos hasta un

nivel de profundidad determinado. Además se ha extendido el sistema de consultas pudiendo hacer consultas sobre los atributos del objeto referenciado.

2.3. Extensión para soportar relaciones de multiplicidad

Como objetivo final se extenderá la librería para soportar relaciones entre objetos que representan multiplicidad. Es decir, permitiremos que un objeto tenga atributos de tipo `List`, `Set` y de tipo `array`.

Al igual que en el objetivo anterior, se tendrá en cuenta la forma de diseñar las tablas que representen la relación para que las operaciones de inserción, actualización, borrado y recuperación sean lo más eficientes posibles

3. Plan de trabajo

Teniendo en cuenta que los dos últimos objetivos del proyecto son incrementales, el plan de trabajo se dividirá en tres fases principales junto a una fase adicional, necesaria para la preparación.

3.1. Primera fase: estudio de las tecnologías existentes

Antes de comenzar con la implementación de la librería hemos dedicado una fase inicial al estudio de tecnologías existentes que nos ayudarán durante el trabajo.

La primera tecnología que analizaremos será la API de reflexión de Java usando el libro Java Reflection in Action [6]. Este análisis nos servirá para conocer los métodos que necesitaremos usar durante la implementación de la librería.

También se revisará la API para ejecución de operaciones sobre bases de datos desde Java denominada JDBC estudiada por los miembros del grupo durante el grado. El libro que hemos utilizado para el estudio es JDBC: Java Database Connectivity [7].

Por último, se estudiarán los patrones de diseño contenidos en el libro Patterns of Enterprise Application Architecture [8] para recurrir a ellos en caso de ser necesario.
Al ser una fase preparatoria, durante la misma también se ha creado el repositorio en la herramienta GitHub en el que se trabajará en las fases posteriores del proyecto.

3.2. Segunda fase: primeros estados de la librería

En esta segunda fase de la librería se procederá a construir la clase principal de la API de la librería a la que tendrá acceso el usuario y a partir de ella implementar los métodos necesarios para operar con la base de datos.

El primer método a implementar será la inserción de objetos. Con los conocimientos adquiridos en la primera fase del plan junto a los diseños razonados para las tablas de la base de datos relacional se creará el método que será el encargado de realizar dicha operación.

Una vez implementada la inserción, y teniendo en cuenta las tablas creadas, se construirá un sistema de consultas para complementar a la librería que permita implementar la recuperación de objetos. Se implementará también un constructor de objetos que se encargue de reconstruir el objeto a partir de los datos obtenidos desde la base de datos.

Después de implementar las operaciones de inserción y recuperación se implementará la actualización y el borrado de objetos teniendo en cuenta que se tendrá que razonar primero sobre la característica de identidad que existe en la orientación a objetos y sobre el modo en que la identidad afecta a dichos métodos.

3.3. Tercera fase: extensión para incluir relaciones entre objetos

Con el fin de extender la API de la librería para soportar las relaciones de un objeto con otros de tipo no básico se estudiarán los cambios que habrá que hacer sobre los métodos ya existentes así como nuevas características.

Para la operación de inserción será necesario razonar sobre la implementación de la relación entre los objetos de cara a construir las tablas de la base de datos relacional.
Se abordarán los problemas surgidos por la creación de las tablas relacionados con la inclusión de los `ID` correspondientes a los atributos de tipo no básico y los problemas a la hora de guardar y recuperar los objetos que sean instancias de clases abstractas.

La funcionalidad de recuperación se tendrá que modificar para incluir soporte de restricciones sobre los objetos de tipo no básico y sobre los atributos de los mismos.
Se tendrá que modificar la construcción de los objetos recuperados afrontando, en caso de que surjan, los problemas derivados de las posibles relaciones circulares entre objetos. También se determinará hasta qué nivel de profundidad se cargarán los objetos recuperados de la base de datos.

Para la operación de actualización se tendrá que modificar el método necesario para incluir la actualización de atributos que no sean de tipo básico.
En la operación de borrado será necesario determinar si se incluirán o no restricciones sobre la relación de los objetos en las tablas de la base de datos relacional.

3.4. Cuarta fase: extensión para incluir relaciones de multiplicidad

Al igual que con la fase anterior, durante esta cuarta etapa se tendrán que estudiar e implementar los cambios necesarios en los métodos para incluir el soporte de relaciones de multiplicidad entre objetos.

En la operación de inserción será necesario pensar la forma de representar dichas relaciones en la base de datos relacional, tanto de tipos básicos como no básicos.

En la operación de recuperación se tendrá que modificar el constructor para soportar este tipo de objetos así como extender el sistema de consultas para permitir restricciones sobre atributos de tipo multivalorado.

En la operación de actualización se tendrá que determinar la forma en la que se tendrán en cuenta las modificaciones hechas sobre los atributos de este tipo.

Capítulo 2: Introduction

1. Background

Object oriented languages still hold an important market share and so do relational databases. Databases use has increased in applications to store information. Traditionally integrated SQL was directly used in the code of the database access layer but this approach is not very sustainable, the reason for that is that the level of abstraction is too low and keeps coupling between the database design and the application. There are several abstraction mechanisms that allow one to unify the object model, used by the programmer, with the relational model, used by the database. This project addresses the integration between the database access layer and the rest of the application. Our objective is to design a library that abstracts the programmer from the database storage approach.

Our library is proposed as an alternative to some of the following existing solutions:

- **Hibernate**: Library used for connecting to a relational database management system. It allows the programmer to set the connection between the object oriented model and the database tables that store those objects, with the help of annotations integrated in the source code,. It also includes a SQL type language named HQL that allows the programmer to make queries on the database to retrieve stored objects.[1]

- **db4o**: Native object oriented data base management system not based on query execution but on the use of library methods to store, retrieve and delete objects. It is not maintained since 2008.[2]

- **ORMLite**: Library similar to Hibernate that also requires annotations on the source code but simpler and more limited. It is oriented to Android platform development.[3]

- **OJB**: Library to map the objects in a relational database using a XML document. It is no longer maintained.[4]

- **ObjectStore**: C++ library that provides a native object oriented database. For the object construction it is necessary to modify the constructor so that the object is stored in the database and memory at the same time. Changes in the object are detected to keep the database updated.[5]

All the technologies mentioned above mostly solve the problem of abstracting the inner details of a database problem. However, we want our library to work on Java and the programmer not to be obliged to use annotations in the code for associating his objects to the tables in the database.

Db4o satisfies these requirements but unfortunately it is no longer maintained. The objective of this project is to build on these ideas to develop a library with a similar interface for the programmer but by using a relational database as a storing mechanism.

2. Objectives

2.1. Object oriented DB library API design

The first and main objective of this project is to design and implement an API for an object oriented database library.

As an underlying storage engine, our library will use a relational database.This choice will be transparent to the programmer, so a different storing engine could have been used (For example: XML or MongoDB).

With this API we want the final user to be able to do every database-related management actions such as insertion, update, deletion or load without having to create the SQL statements himself for those actions.

To do the insertion, update and deletion operations the user of the library will have at his disposal methods of the library corresponding to each process. These methods will receive, as a method parameter, the object to insert, update or delete, without having to provide information about the tables or columns of the database involved in the operation.
A query system with library-supported constraints supported will be designed for obtaining objects from the database. Queries will return already constructed objects using the corresponding rows in the database.

As a first objective, only objects with basic type attributes (integers, strings, etc.) will be used at the beginning. The necessary mechanisms for the management of object identities will be implemented, so, for example, if an object is saved twice or more, a single representation will be stored on the database.

2.2. Extension to support relations between objects

To improve the insertion, update, deletion and retrieval of objects mentioned in the previous objective, the API will be extended to support objects with references to non-basic type objects, except container type objects like lists, sets or arrays.

To implement this objective successfully we will study the right way to construct the SQL tables in the relational database in order to represent the relations between objects and to devise mechanisms for retrieving references chains up to a given depth level. The query system will also be extended so that the user can make queries on the attributes of the referenced object.

2.3. Extension to support multiplicity relations

As a final objective the library will be extended to support relations between objects that represent multiplicity. In other words, the library will allow an object to have `List`, `Set` and `array` type attributes.

As in the previous objective, we will study the way of designing the tables that represent the relation, so the insertion, update, deletion and retrieve operation are as efficient as possible.

3. Workplan

Having in mind that the two later objectives are incremental, the workplan will be divided into three main phases along with a previous phase, necessary to prepare the project.

3.1. Phase one: study of the existent technologies

Before we start with the implementation of the library we have dedicated an initial phase to study the existing technologies that will help us during the work.

The first technology that we will analyze is the Java reflection API by using the book titled Reflection in Action [6]. This analysis will help us to know the methods that we shall use during the implementation of the library.

We will also overhaul the API for the execution of operations on databases by using Java JDBC, which had been studied by the members of the group during the degree. The book used for this study is JDBC: Java Database Connectivity [7].

Finally, we will study the design patterns from the book Patterns of Enterprise Application Architecture [8] to use them in case they are necessary.
Considering that this is a preparatory phase, during it we have also created a GitHub repository to work on in the next phases of the project.

3.2. Phase two: library first stage

In this second phase of the project we will proceed to create the main class of the library API that will be instantiated by the user. In this class we will implement the methods needed to operate over the database.

The first method to implement will be the insertion of objects. With the knowledge acquired during the first phase of the work plan and the already developed design of the database tables, the method in charge of doing this operation will be implemented.

Once the insertion is implemented and the tables are created, we will build the query system to complement the library that allows it to retrieve objects from the database. A constructor in charge of rebuilding the objects from the data retrieved from the database will also be implemented.

After implementing the insertion and retrieval operations we will implement the update and deletion of objects having in mind the identity property that exists in object oriented languages and the way it affects the above mentioned methods.

3.3. Phase three: extension to include relations between objects

In order to extend the API of the library to support relations between objects with others non-basic type objects, we will study the changes that will be needed to make in the existing methods and also how to implement the new features.

For the insertion operation it will be necessary to study the implementation of the relation between objects so we can construct the tables of the relational database.
All the problems encountered related to inclusion of the `ID` belonging to the non-basic type objects on the tables and the storing of abstract class instances will be resolved.

The retrieval operation will be modified to include the support of constraints over the non-basic type objects and its attributes.
The object construction will also need modifications to solve all the problems related to circular relations between objects.
We will also determine the depth level up to which the objects will be retrieved.

For the update operation we will have to make the necessary changes to include the update of non-basic type attributes.
In the deletion operation we will need to determine if restrictions over object relations in the database tables will be included or not.

3.4. Phase four: extension to support multiplicity relations

As in the previous phase, during this fourth phase we will need to study and implement the necessary changes in the methods to include the support of multiplicity relations between objects.

In the insertion operation it will be necessary to study the way of representing the above mentioned relations in the relational database, for non-basic and basic objects.

In the retrieval operation we will need to change the object constructor, so it supports this type of objects and we will do the same with the query system, so it supports constraints over multivalued attributes.

In the update operation we will need to determine the way the changes in this kind of attributes will be implemented.

Capítulo 3: Selección de herramientas y tecnologías

Dado que este proyecto consiste en el desarrollo de una librería para la gestión de bases de datos nativas orientadas a objetos en Java, hemos optado por seleccionar las siguientes herramientas y tecnologías:

1. API de reflexión de Java

La reflexión es una característica de algunos lenguajes de programación que permite acceder, en tiempo de ejecución, a sus componentes de alto nivel como clases, atributos o métodos. Dado que nuestra librería trabaja con objetos que pueden ser instancias de cualquier clase, hemos de saber a qué clase concreta pertenece cada objeto para poder determinar qué atributos tiene cada uno para guardarlo en la base de datos. Del mismo modo, también se necesita reflexión para poder reconstruir un objeto a partir su información en la base de datos.

En nuestro proyecto, las principales clases relacionadas con la reflexión de Java que hemos utilizado han sido la clase `Class` y la clase `Field`.

La clase `Class` representa la clase a la que pertenece una instancia de un objeto. Mediante los métodos `getName()` y `getFields()` hemos podido acceder al nombre de la clase y a sus campos.

La clase `Field` representa cada uno de los campos pertenecientes a una clase de Java. Permite acceder a su nombre mediante el método `getName()`, a su clase usando el método `getType()`, a su valor asociado en una instancia del objeto a través del método `get(Object obj)` y modificar su valor en una instancia del objeto mediante `set(Object obj, Object value)`.

Supongamos una instancia de una clase denominada `Prueba`, incluida en el paquete `ucm.fdi`, que contiene dos atributos: `str` de tipo `String` y `num` de tipo `int` (Diagrama de clases: Figura 3.1). Esta instancia se ha creado con nombre `prueba` y con los valores `"cadena"` para `str` y `1` para `num`.

Figura 3.1

Si utilizamos el método `getClass()` de nuestra instancia obtendremos un objeto de tipo `Class<?>` que contendrá la clase asociada a la instancia prueba. Este objeto lo asociamos a la variable `clasePrueba`.
Sobre esta nueva variable podemos ejecutar los métodos nombrados anteriormente de la clase `Class`. Si ejecutamos el método `getName()` obtendremos un `String` que contendrá `"ucm.fdi.Prueba"` y si ejecutamos el método `getFields()` obtendremos un array de atributos de tipo `Field` con dos posiciones que corresponden a las variables `str` y `num`, en ese orden.

Si recorremos este array podremos acceder a los campos y obtener su valor, modificarlo u obtener su tipo. Por ejemplo, si para la posición [0] del array, que corresponde a la variable de tipo `String`, ejecutamos el método `getName()` obtendremos `"str"`, y si ejecutamos el método `getClass()` obtendremos un objeto de tipo `Class` que corresponde a la clase `String`.

Si queremos obtener el valor asociado a ese campo tendremos que especificar para qué instancia queremos obtener dicho valor (en nuestro caso el objeto `prueba`) dentro del método `get` de la forma `getFields()[0].get(prueba)`. Con esto, obtendremos el valor de nuestro campo `str` que, como se especificó anteriormente, es `"cadena"`.

Para modificar el valor, al igual que para obtenerlo, hay que especificar el objeto del que se quiere modificar, además del nuevo valor, utilizando el método `set` de la forma `getFields()[0].set(prueba,"cadena2")`. Después de esto, si ahora obtenemos el valor del campo `str` el resultado será `"cadena2"`, en vez que cadena, como obtuvimos anteriormente.

2. JDBC (Java Database Connectivity)

Es una librería que sirve para conectar con varios gestores de BD desde una aplicación Java y poder ejecutar sentencias SQL desde la misma. Como nuestro objetivo es el desarrollo de una librería en Java que trabaje con bases de datos, estas dos características de JDBC han sido importantes para trabajar. Con esta librería se pueden usar consultas paramétricas. Estas consultas previenen la inyección SQL, es decir, sirve para que el usuario de la aplicación no introduzca código malicioso en las entradas de las operaciones sobre una base de datos. Esto se hace de forma que al construir la consulta SQL en vez de introducir el valor previsto se introducirá un marcador (placeholder) que en nuestro caso será una interrogación. Después se añadirá el valor correspondiente mediante los métodos de la clase `PreparedStatement`.

Poniendo un ejemplo relacionado con nuestro proyecto.

1. Primero nos construimos la sentencia SQL. En vez de introducir el valor, introduciremos un marcador.

```
String sql = "SELECT nombreclase,nombretabla FROM indicetabla
WHERE   nombreclase = ?"
```

2. Nos crearemos un objeto de la clase `PreparedStatement`.

```
PreparedStatement pst;
```

3. Pasando la consulta mediante un método a nuestra conexión en la base de datos nos devolverá un objeto de `PreparedStatement`.

```
pst = c.prepareStatement(sql);
```

4. Como en este caso es un `String` llamaremos al método `setString` del `PreparedStatement`, En los valores el primero será 1 por qué es el primer marcador y el segundo será el valor que queremos introducir.

```
pst.setString(1, nombreClase);
```

3. Maven

Es una herramienta de gestión de proyectos. Permite, por un lado, ejecutar las tareas habituales de desarrollo. Pero, por otro lado, permite gestionar las librerías. Esta tecnología nos ha permitido incluir las distintas librerías (c3p0, mysql-conector-java, commons-lang3, commons-io) utilizadas en nuestro proyecto sin necesidad de descargar manualmente una a una. Esto lo hemos hecho añadiendo al archivo pom.xml en el apartado de dependencias (Figura 3.2).

```
<dependencies>
  <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>5.1.20</version>
  </dependency>
  <dependency>
      <groupId>org.apache.commons</groupId>
      <artifactId>commons-io</artifactId>
      <version>1.3.2</version>
  </dependency>
  <dependency>
      <groupId>com.mchange</groupId>
      <artifactId>c3p0</artifactId>
      <version>0.9.5</version>
  </dependency>
  <dependency>
      <groupId>org.apache.commons</groupId>
      <artifactId>commons-lang3</artifactId>
      <version>3.3.2</version>
  </dependency>
</dependencies>
```

Figura 3.2

4. Github

Git es una herramienta usada, principalmente, en desarrollo de software que facilita el control de las versiones de un proyecto mediante repositorios a los que se accede mediante órdenes o comandos.

GitHub una plataforma web (Figura 3.3) que usa la tecnología Git descrita anteriormente para guardar, en la nube, proyectos, sus versiones, cambios de cada versión y contribuciones de cada miembro del equipo. En GitHub hemos guardado nuestro proyecto

para tener más control sobre las versiones y contribuciones individuales al proyecto, y poder disponer siempre de la última versión desde cualquier equipo.

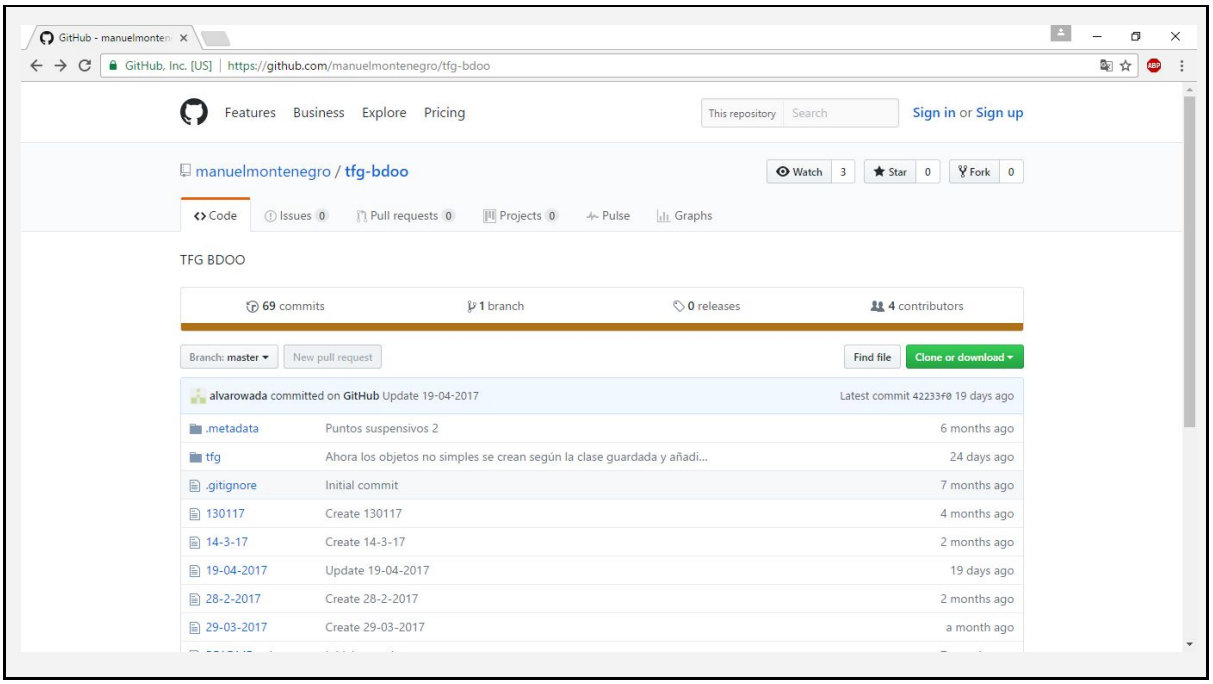

Figura 3.3

5. Git Shell

Consola de comandos (Figura 3.4) que permite la ejecución de comandos para acceder y hacer modificaciones en un repositorio de git.

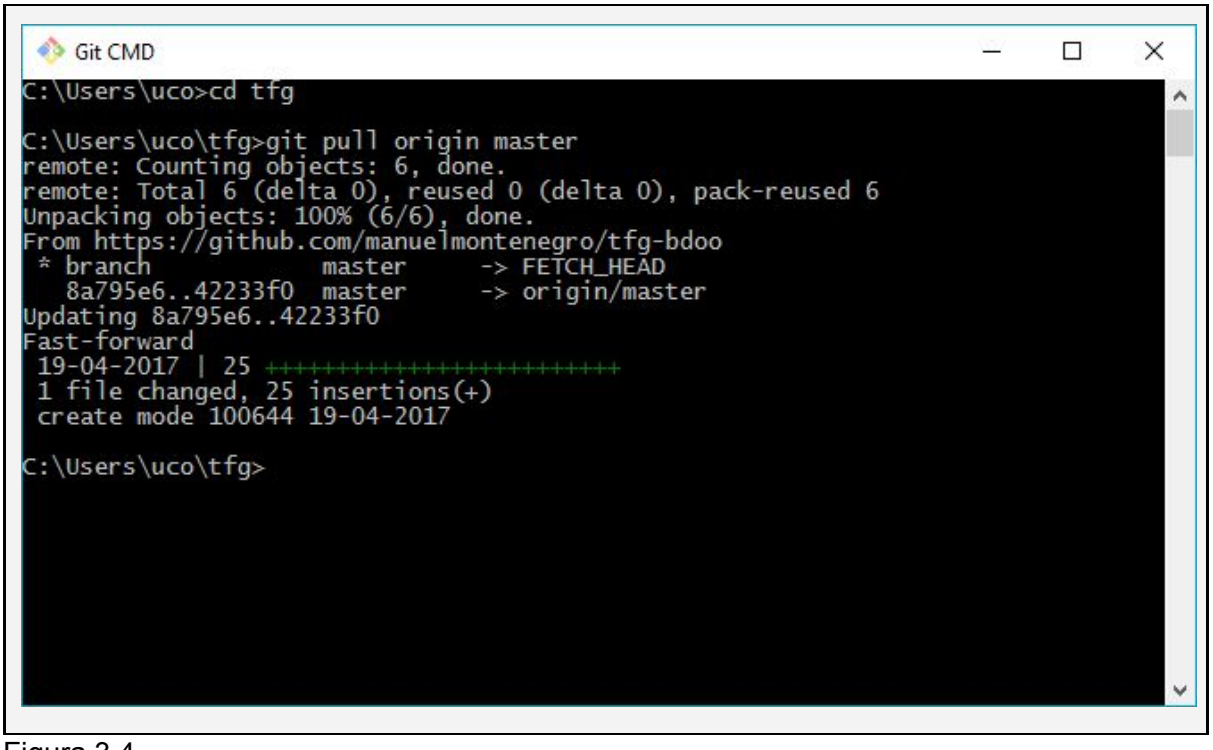

Figura 3.4

6. Eclipse

Es un Entorno Integrado de Desarrollo (IDE) que sirve, entre otras cosas, para desarrollar aplicaciones Java (Figura 3.5). Dado que todos los integrantes del grupo hemos trabajado anteriormente con Eclipse, nos ha parecido adecuado utilizarlo para desarrollar la librería.

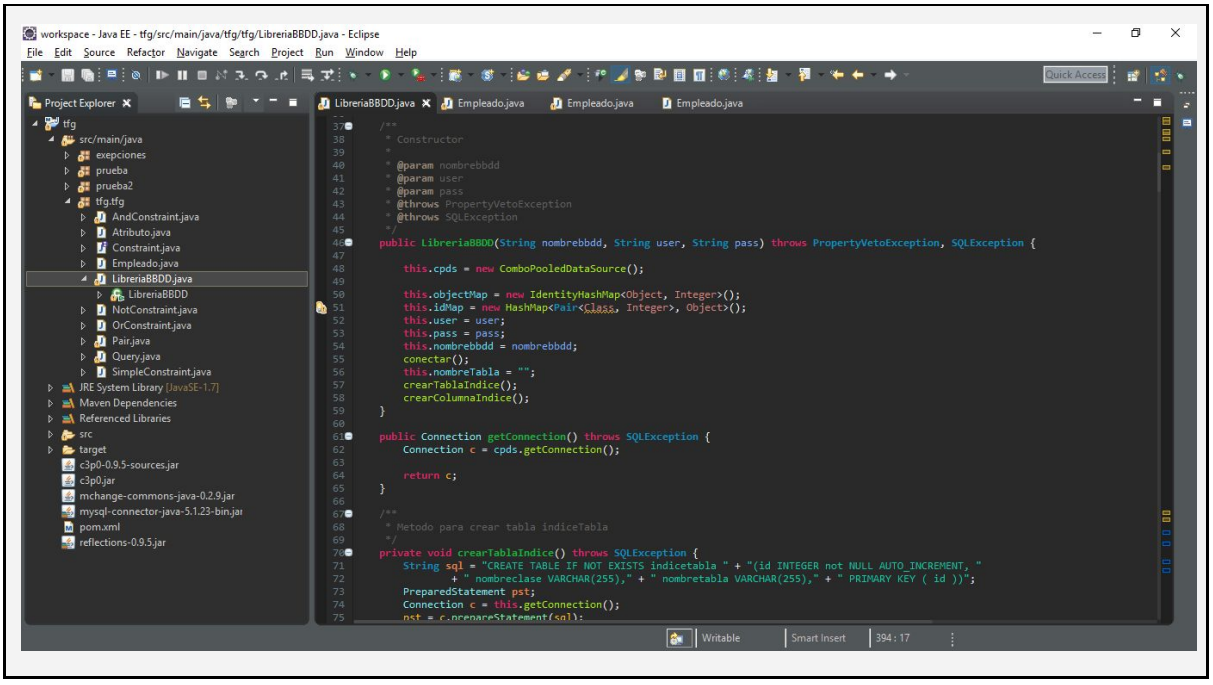

Figura 3.5

7. Github for Eclipse

Plugin para el entorno Eclipse que permite gestionar el repositorio del proyecto desde el mismo Eclipse, sin necesidad de ejecutar comandos desde la consola.

8. XAMPP

Es un paquete que incluye Apache, MySQL y PHP. Sus principales funciones son el gestor de bases de datos MySQL, el servidor web Apache y los intérpretes para lenguajes de script: PHP y Perl. Nosotros hemos usado Apache y MySQL para gestionar la base de datos en phpMyAdmin. A continuación se detalla cada una de estas tres tecnologías:

- **MySQL**
  Es un servidor que gestiona bases de datos relacionales, permitiendo la ejecución de consultas SQL. Como las bases de datos gestionadas por nuestra librería se almacenan internamente en una BD relacional, necesitamos este SGBD para poder almacenar los datos y ejecutar nuestras consultas. De todas las tecnologías nombradas en este capítulo esta es la única que utilizará el usuario final.

- **phpMyAdmin.**
  Ofrece un entorno web para conectar con una base de datos y así poder crearnos la base de datos con sus correspondientes gestiones, creación de tablas, consultas sobre su contenido, etc, y ademas, ejecución de sentencias en la consola SQL . Durante el desarrollo de nuestra librería, utilizamos este entorno para comprobar el estado de las diferentes tablas de nuestras bases de datos. aunque el usuario final al que nuestra librería está destinado no necesita este entorno salvo para crear la base de datos. La interfaz de phpMyAdmin se muestra en la siguiente figura (Figura 3.6).

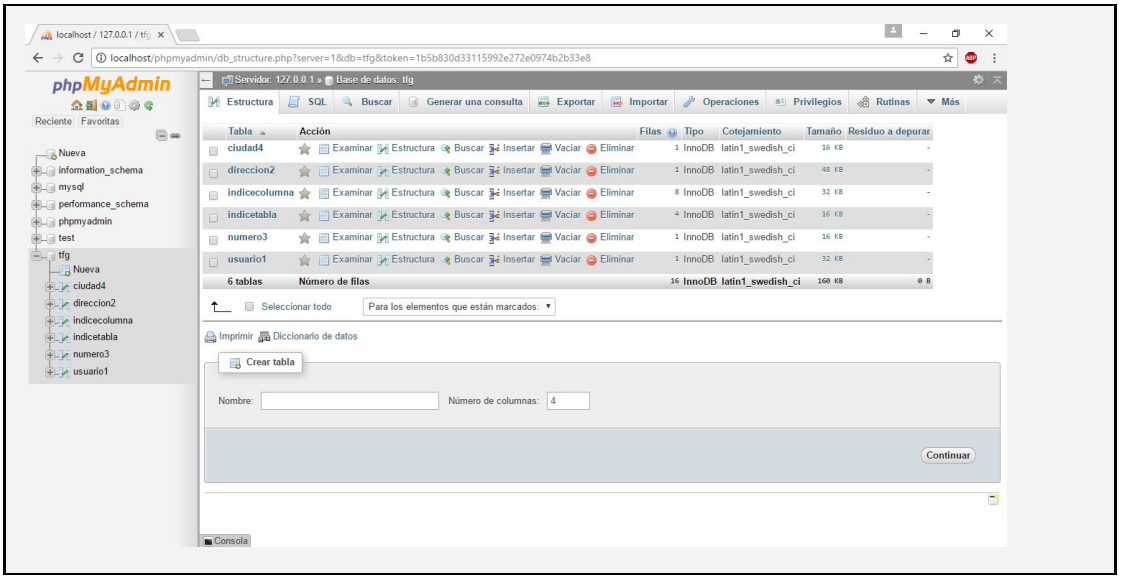

Figura 3.6

- **Apache**
  Servidor web HTTP para procesar peticiones de páginas web. El entorno web phpMyAdmin se ejecuta sobre este servidor web.

Capítulo 4: Primera fase: objetos con atributos simples

Para facilitar las primeras fases del desarrollo de la librería se ha decidido que, en esta primera fase, únicamente se trabajará con objetos que contengan atributos con tipo básico de Java. En los primeros ejemplos de objetos los atributos tratados han sido `String` e `int`. Se ha decidido usar estos dos por la disparidad de funcionalidad entre ambos y por su habitualidad. Posteriormente se ha extendido a todos los atributos básicos de Java.

Durante el desarrollo de este apartado se recurrirá repetidamente al siguiente ejemplo de objeto: Supongamos una instancia de una clase denominada `Empleado` (diagrama de clase: Figura 4.1) incluida en el paquete `tfg`, que a su vez se encuentra dentro de otro paquete con el nombre `ucm`. Esta clase tiene dos atributos simples: `nombre`, de tipo `String`, y `edad`, de tipo `int`. Esta instancia se ha construido con los valores nombre `"Manuel"` y edad `29`.

Figura 4.1

1. Índices para tablas

Para la organización de las tablas correspondientes a cada una de las clases de los objetos que se guardarán en la base de datos, se ha implementado una tabla índice que contendrá la información de cada una de las otras tablas.
Esta tabla índice, en caso de no existir previamente, se crea con el nombre `indicetabla` al llamar al constructor de la librería. Las columnas de esta tabla son:

- `ID`: identificador de cada fila. Es la clave primaria de la tabla.

- `NombreClase`: el nombre completo, incluyendo los paquetes, de la clase del objeto que se ha insertado en la base de datos. Si existieran varias instancias de una misma clase sólo será necesario introducir la primera vez su nombre completo, y este valdrá para las siguientes instancias.

- `NombreTabla`: el nombre de la tabla donde se almacenarán los objetos correspondientes a la clase almacenada en `NombreClase`. El nombre de la tabla se ha construido con la unión del identificador de la propia fila en la tabla de índices más nombre de la clase (sin incluir el paquete) . De esta forma se asegura que dos clases con el mismo nombre, pero situadas en distintos

paquetes, no entren en conflicto a la hora de asignar los nombres de sus respectivas tablas.

Aplicando esto al ejemplo nombrado anteriormente, al insertar la instancia de la clase Empleado se insertará también en la tabla de índices una fila con los valores: `1` para el identificador de fila (asumiendo que es la primera inserción usando la librería), `ucm.tfg.Empleado` para el nombre de la clase y como nombre de tabla `1Empleado`.

ID	NombreClase	NombreTabla
1	ucm.tfg.Empleado	1Empleado

2. Índice de columnas

Como complemento al índice de tablas también hay un índice que guarda, para cada atributo de los objetos, su nombre y el nombre que va a tener su columna correspondiente en la base de datos. Aunque ambos nombres son iguales, esta fila se inserta para llevar cuenta de los atributos para los que ya se ha creado su columna correspondiente.
Esta tabla índice se crea con el nombre `indicecolumna` al llamar al constructor de la librería. Las columnas de la tabla son:

- `ID`: identificador de cada fila. Es la clave primaria de la tabla.
- `idTabla`: Hace referencia al ID de la tabla `indicetabla` donde se ha guardado el nombre de la clase del objeto. Esta columna es necesaria ya que este este índice es común para todos los objetos y por lo tanto se necesita diferenciar las distintas columnas a qué tabla pertenecen.
- `atributo`: Se guarda el nombre del atributo del objeto.
- `columna`: Se guardará el nombre de la columna de la tabla del objeto que va a tener el atributo correspondiente.

Teniendo esto en cuenta usando el mismo ejemplo anterior, al insertar al empleado se inserta también en esta tabla dos filas con los valores siguientes:

ID	idTabla	Atributo	Columna
1	1	nombre	nombre
2	1	edad	edad

3. Identity Map

En programación orientada a objetos dos objetos pueden ser distinguibles a pesar de que contenga la misma información. Esto es lo que se comprueba en Java con el operador !=. Sin embargo, en una tabla SQL dos filas son iguales si contienen exactamente la misma información (de ahí que añadamos un `ID` numérico a cada tabla, para poder distinguir dos objetos que tengan la misma información). Además de los motivos de eficiencia, este Identity Map nos permite conocer qué instancias están vinculadas a la base de datos, de modo que se pueda hacer `update()` o `delete()` sobre ellas. Tenemos dos atributos en la clase principal que nos ayudan en estos principios.

Uno de ellos se llama `objectMap` y consiste en un `IdentityHashMap` que guarda la dirección de memoria el objeto como clave y el `ID` de la fila en la tabla de la base de datos en la que se ha insertado el objeto como valor. Necesitamos que sea un `IdentityHashMap` ya que queremos comparar el objeto con el == en vez de con el `equals()` característico de los `HashMap`.

Usamos este atributo en el método para insertar un objeto en la base de datos. Si el objeto que estamos intentando guardar esta contenido en el `identityHashMap` no hace falta guardarlo, ya que el hecho de que esté contenido en este `objectMap` quiere decir que ya se ha insertado anteriormente, debido a que al insertar un objeto guardamos en el `objectMap` el objeto y el `ID` de la fila en la base de datos.

Por otro lado se encuentra el atributo llamado `idMap`. Se trata de un `HashMap` dado que en este caso necesitamos comparar con el método `equals`. Esta tabla guardará como clave una clase llamada `Identificador` la cual contiene un Integer (`ID` de la fila de la tabla de la base de datos) y un `String` (nombre de la ruta de la clase del objeto). Como valor del `HashMap` tendremos el objeto que queremos recuperar. Aplicándolo a nuestro ejemplo este Identificador será la clave de nuestro `HashMap` y contendrá como `String` "`ucm.tfg.Empleado`" y como `Integer` el valor `1` .

Usamos este atributo en el método `executeQuery`. Si el objeto que estamos intentando crear se encuentra en nuestro `idMap` no hace falta que lo creemos, ya que podemos recuperarlo directamente de este `HashMap`. Para saber si está contenido necesitamos el nombre cualificado (es decir, incluyendo el paquete) de la clase del objeto y su `ID`. Con estos dos datos construimos el objeto `Identificador` y podremos ver si está contenido en el `idMap`. En caso de que no lo esté necesitaremos crear el objeto a partir del `ResultSet` y añadirlo al `idMap` y al `objectMap`. En el método de guardar, insertamos en el `idMap` el objeto guardado en la base de datos como valor y como clave el `ID` de la fila guardada y el nombre de la ruta de la clase del objeto.

4. Inserción, actualización borrado y consultas

4.1. Inserción

Para la inserción en la base de datos lo primero que haremos será recorrer los atributos del objeto y para cada uno guardar su nombre y el tipo SQL que deberemos tener para guardar el atributo en la base de datos. En caso de que sea un `String` le adjudicamos un tamaño de 255. Para guardar esto usamos una lista con objetos de una clase que hemos creado, llamada `Attribute`, que tiene dos atributos de tipo `String`: uno para el nombre y otro para el tipo. Continuando con nuestro ejemplo esta lista contendrá dos atributos.

Atributos clase Atributo	**Atributo 1**	**Atributo 2**
nombre	nombre	edad
tipo	VARCHAR(255)	INTEGER

Con esta lista de atributos y el nombre de la clase crearemos la tabla correspondiente en la base de datos solo si no existe. Antes tendremos que insertar esta clase en la tabla `indicetabla` explicada anteriormente. Para ello necesitamos el nombre de la clase que ya hemos recuperado previamente y nos falta el nombre que va a tener la tabla en la base de datos. Para saber este nombre, lo primero que hacemos es buscar en `indicetabla` si ya existe el nombre cualificado (esto es, incluyendo paquete) de la clase, que es único. Si existe no hará falta insertarlo de nuevo, pero si no existe cogeremos el máximo `ID` de la tabla y le sumaremos uno para obtener el número que debemos añadir al nombre de la clase para crear una tabla con ese nombre.

Una vez insertada la fila en `indicetabla`, crearemos la tabla con el nombre que hemos obtenido previamente, con la condición de crearla solo si no existe ya, puesto que si la creásemos de nuevo perderiamos todos los datos guardados. Inicialmente esta tabla contendrá una única columna: la clave de la tabla.

Después recorreremos la lista de atributos y para cada uno de ellos miramos en la tabla `indicecolumna` si ya existe ese atributo en esa tabla. En caso de que no exista, lo guardaremos. Para ello necesitaremos el `ID` de la fila donde se encuentra nuestra tabla en el `indicetabla` y el nombre de cada atributo. También debemos alterar la tabla que contendrá los datos del objeto a insertar, ya que deberemos añadir las columnas correspondientes a los atributos Para esto nos sirve el atributo tipo de la clase `Attribute`.

Por otra parte, borraremos los atributos que ya no estén en la clase, quitandolos también de la tabla `indicecolumna`.

Posteriormente comprobaremos si el objeto está contenido en el `Identityhashtable` *`objectMap`* debido a que de así sea, querrá decir que ya le hemos insertado anteriormente o recuperado de la base de datos por lo tanto no deberemos insertarlo en la base de datos y lanzaremos una excepción de `DuplicatedObject`. Lo que haremos es sacar los atributos del objeto que no sean nulos para guardar en la base de datos solo esos atributos, estos los meteremos en una lista de `Attribute`, la cual nos recorreremos cogiendo el nombre de cada atributo y consiguiendo su valor para y así guardarlo en la tabla correspondiente de la base de datos.

Después de haber insertado obtendremos el `ID` de la última inserción ya que le deberemos devolver para crearnos un `Identificador` con ese `ID` y el nombre de la clase para guardarle en el `hashTable idMap`. También guardaremos el objeto como clave en el *`objectMap`* y como valor el `ID` que acabamos de devolver.

4.2. Actualización

El método `update` modificará la fila donde están los datos del objeto que se recibe, para de esta forma actualizar la base de datos con los posibles cambios que el objeto haya sufrido.

Para la actualización de objetos de la base de datos, lo primero es comprobar que el objeto que se recibe como parametro del metodo `update` esté en el `objectMap`, por que solo se pueden actualizar objetos que realmente estén en la base de datos, además es necesario que esté en los mapas para poder saber cual es el identificador de la fila que queremos actualizar.

En el caso de que el objeto ya haya sido guardado o recuperado de la base de datos previamente y que por lo tanto ya está en los dos mapas y la fila que queremos actualizar ya exista, se hace una consulta al índice de tablas para saber cual es el nombre de la tabla donde están los datos de ese objeto, se recorre los atributos del objeto para ir guardando su nombre y valor actual, y por último se consulta en el `objectMap` cual es `ID` de su fila que se va a actualizar. Con estos datos se construye una sentencia SQL para actualizar esa fila en concreto con los datos actuales del objeto.

Poniendo como ejemplo nuestra instancia de la clase *`Empleado`*, una vez guardada y cambiado su atributo `edad` por `27`, realizaremos una actualización del mismo en la base de datos. Como ya ha sido guardado anteriormente estará en el `objectMap` (ver sección 4.3) y, por tanto, podremos actualizarlo sin ningún inconveniente ya que en esta tabla hash tendremos el `ID` de la fila en la que se encuentra el `Empleado`.

1. Procederemos a recorrer los atributos no nulos de nuestro `Empleado` para guardarlos en una lista de `Attribute`.
2. Recorreremos esta última lista para ir formando la sentencia SQL:

a. Primero generaremos la cadena que indica las modificaciones a realizar. Esta cadena será de la forma:

   nombre = ? , edad = ?

b. En una lista de objetos llamada `valores` guardaremos los valores que vamos a introducir en los marcadores de la cadena anterior, los cuales sacaremos de la siguiente forma: crearemos un objeto de la clase `Field`(siendo `o` nuestro `Empleado`) y en `valores` guardaremos el valor de `val`.

```
Field val = o.getClass().getDeclaredField(atributo.getNombre());
valores.add(val.get(o));
```

3. Después buscaremos en la tabla `indiceTabla` el nombre de la tabla donde tenemos que hacer la actualización. Esto nos devolverá `1Empleado`.
4. Con todos estos datos construimos una consulta SQL paramétrica de la forma:

   Update set Empleado1 set nombre = ?, edad = ? where id = ?

5. Por último tendremos que rellenar los tres marcadores de la consulta paramétrica:
   a. Los dos primeros los tendremos en la lista de `valores` `(Manuel,27)`.
   b. El tercero es la clave del objeto, que conseguimos a partir del `Identity Map`.

4.3. Borrado

El método `delete` elimina de la base de datos la fila con todos los datos del objeto que reciba como parámetro.

Para el borrado de los datos del objeto lo primero es comprobar que el objeto que se recibe como parametro del metodo `delete` esté en el `objectMap`, por que solo se pueden borrar objetos que se hayan guardado o recuperado de la base de datos previamente, ya que de otra forma no estaría en ninguno de los mapas y no podría averiguarse su `ID` de forma inequívoca. En esta caso el método terminará con la excepción `NonExistentObject`.

En el caso de que el objeto ya haya sido guardado o recuperado de la base de datos previamente y que por lo tanto ya está en los dos mapas y su tabla correspondiente creada y añadida al índice de tablas, se hace una consulta al índice de tablas para saber cual es el nombre de la tabla donde se guardan los objetos de esa clase, con esto y con su `ID`, que se puede recuperar del `objectMap`, se construye la sentencia SQL que se encarga de borrar los datos de este objeto en particular. Después de haber borrado los datos del objeto de la base de datos se elimina su entrada de los dos mapas.

4.4. Consultas

4.4.1. Recuperación mediante restricciones

Para la recuperación de objetos de la base de datos se ha decidido implementar una versión del patrón Query Object, incluido en el libro Patterns of Enterprise Application Architecture [8].

A la hora de realizar una consulta en la base de datos, el usuario de nuestra librería ha de indicar qué objetos quiere recuperar. Por ello, antes de lanzar la consulta, este ha de indicar unas restricciones de consulta. En esta implementación las restricciones de la consulta implementan una interfaz denominada `Constraint`. Las instancias que se pueden crear de esta interfaz son las siguientes:

- *SimpleConstraint*: se corresponde con un operador relacional aplicado a dos operandos: un nombre de atributo y un valor. Los operadores relacionales soportados son la igualdad, incluyendo además los operadores de menor (<), mayor (>), menor o igual (<=), mayor o igual (>=) y diferencia (<>).

- *NotConstraint*: representación de la negación de otra restricción. La implementación contiene una *Constraint* a la que se aplicará la negación.

- *AndConstraint*: representa la restricción de cumplimiento simultáneo de dos o más restricciones (operador lógico *AND*). Se ha implementado como una lista de instancias de la interfaz *Constraint*.

- *OrConstraint*: representa la restricción de cumplimiento de al menos una de dos o más restricciones (operador lógico *OR*). Al igual que la *AndConstraint*, la implementación es mediante una lista de instancias de la interfaz *Constraint*.

Esta implementación se corresponde con el siguiente diagrama de clases (Figura 4.2):

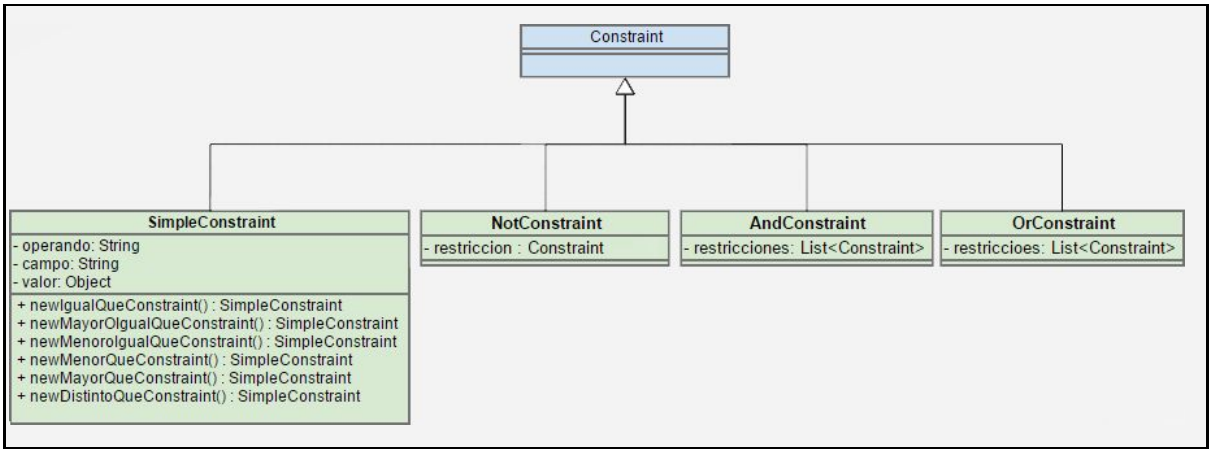

Figura 4.2

Para completar la implementación de la recuperación de objetos se ha creado la clase `Query`, que será la encargada de construir la sentencia SQL con las restricciones a aplicar, incluidas como una instancia de la interfaz `Constraint` como atributo de la clase, y ejecutarla. Estas dos operaciones se han codificado en los siguientes métodos:

- `toSql`: este método se encarga de analizar la restricción que se aplicará y construirá la sentencia SELECT correspondiente. Devuelve la sentencia en forma de `String`.

- `executeQuery`: es el método encargado de adquirir una conexión con la base de datos de la librería, obtener la sentencia SQL del método descrito anteriormente y ejecutarla. Devolverá una lista de objetos que cumplen las restricciones aplicadas.

Para construir cada uno de los objetos que se reciben como fila de una tabla de la base de datos la clase `Query` se apoya en otra clase externa, `ObjectCreator`.
Esta clase contiene el método `createObject`, que recibe el `ResultSet` obtenido de la ejecución de la sentencia y la clase correspondiente al objeto que se tiene que crear en forma de instancia de la clase de Java `Class`.

La forma que tiene el método de construir el objeto es la siguiente:
En primer lugar, usando la clase `Class` de la API de reflexión de Java, se construye una nueva instancia de la clase del objeto que se devolverá.
Lo segundo será obtener los campos de la clase, que se reciben en forma de `array` de `Field`, clase que permite analizar y asignar valores a un campo de un objeto.
Por último, para cada uno de los campos obtenidos, se conseguirá su correspondiente valor del `ResultSet` y se asignará al campo usando el método `set` de la clase `Field`.

Este proceso de creación se ejecutará para cada una de las filas obtenidas en el `ResultSet` y añadirá los objetos a una lista de `Object`, que será la que devuelva el método `executeQuery`.

Si queremos aplicar esta forma de recuperación al ejemplo que se ha desarrollado en este apartado (recuperación de un `Empleado` con `nombre` “`Manuel`” y `edad 29`) se hará de la siguiente forma:

1. Creación de la instancia de la clase `Query` que se encargará de obtener el objeto. El atributo `lib` utilizado en el constructor es el objeto creado de la clase `LibreriaBBDD`.
   ```
   Query q = lib.newQuery(Empleado.class, lib);
   ```

2. Creación de cada una de las restricciones individuales usando la clase `SimpleConstraint`.
   ```
   SimpleConstraint c1 = SimpleConstraint.newEqualConstraint("nombre","Manuel");
   SimpleConstraint c2 = SimpleConstraint.newEqualConstraint("edad",29);
   ```

3. Creación de la restricción AND con las dos restricciones creadas anteriormente.
   ```
   AndConstraint c = new AndConstraint(c1,c2);
   ```

4. Asignación de la restricción creada en el punto 3 a la instancia de la clase `Query` creada en el punto 1.
   ```
   q.setConstraint(c);
   ```

5. Ejecución de la consulta y obtención de la lista de objetos.
   ```
   List<Empleado> l = (List<Empleado>) q.executeQuery();
   Empleado manuel = l.get(0);
   ```

En el último punto del proceso descrito, el método `executeQuery` es el encargado de construir y ejecutar la sentencia SQL para recuperar el objeto.
Para explicar la construcción de la sentencia vamos a apoyarnos, de nuevo, en el ejemplo anterior y simular una variable de tipo `String` que será el resultado de la composición. El nombre de esta variable es `sqlStatement`.

Lo primero que tenemos que obtener es la tabla correspondiente a la clase del objeto. Como la clase la conocemos (es una de las variables de la clase `Query`) tan solo tenemos que obtener su tabla usando el método `getTableName` de la librería. Este método consulta el `Map` que almacena las correspondencias entre las clases y las tablas. Si se encuentra en el mapa lo devolverá y si no lo está se hará una consulta a la BD para conocer el nombre de la tabla y se añadirá al mapa por si es necesario utilizarlo más adelante.
Con esto comenzamos a construir la sentencia de la siguiente forma:

```
sqlStatement = "SELECT * FROM " + lib.getTableName(clase.getName()) + " WHERE
";
```

Una vez hemos llegado al estado anterior de la sentencia lo único que nos queda es añadir la restricción o restricciones asignadas a la consulta. Para obtener la restricción en formato sql dentro de la clase `Constraint` se incluye el método `toSql`.
En el caso de las restricciones `SimpleConstraint` el método `toSql` devolverá el nombre del campo, su operador y un signo de interrogación. El signo de interrogación posteriormente se sustituirá por el valor utilizando el método `setObject` de la clase `PreparedStatement`. Se ha implementado de esta forma para asegurar que se formatea de forma correcta la igualdad y evitar inyección SQL.
Si la restricción es una AND o una OR el método `toSql` devolverá la unión del resultado de cada una de las restricciones de sus correspondientes listas intercalando el operador correspondiente (AND/OR) entre medias.
El último caso es que la restricción sea de tipo NOT. En esta ocasión se devolverá el resultado de la restricción asignada con NOT delante.

Teniendo esto en cuenta, los métodos `toSql` de las restricciones del ejemplo (c1, c2, y c) devolverán lo siguiente:

c1: nombre = ?
c2: edad = ?
c: nombre = ? AND edad = ?

Habiendo obtenido las restricciones en este formato solo nos queda añadirlas a nuestra variable para completar nuestra sentencia:

```
sqlStatement += restriccion.toSql();
```

El resultado, por tanto, de la variable que se devolverá en el ejemplo es el siguiente:

```
SELECT * FROM Empleado1 WHERE nombre = ? AND edad = ?
```

Teniendo esta sentencia lo único que tenemos que hacer antes de ejecutarla es asignar los campos de las restricciones para sustituir los marcadores de la consulta paramétrica. Para hacer esto cada restricción tiene un método llamado `getValues`, que devolverá una lista de `Object` con cada uno de los valores en orden en el que se deben asignar. En el ejemplo anterior, cada uno de los métodos `getValues` de las restricciones devolverán:

c1: [manuel]
c2: [29]
c: [manuel, 29]

Una vez hemos obtenido la lista de la restricción `c` solo tenemos que recorrerla y, como hemos mencionado anteriormente, usar el método `setObject` de la clase `PreparedStatement` para asignar cada valor donde corresponde, dando como resultado nuestra sentencia SQL final lista para ejecutar:

```
SELECT * FROM 1Empleado WHERE nombre = “manuel” AND edad = 29
```

Tras obtener la sentencia y ejecutarla obtendremos un objeto de la clase `ResultSet`, que contendrá las filas recuperadas de la base de datos que cumplen las restricciones. En nuestro caso, suponiendo que en nuestra base de datos solo hemos guardado el objeto del ejemplo, el `ResultSet` contiene:

ID	**nombre**	**edad**
1	manuel	29

Este set de resultados obtenido es el que recibirá el constructor de objetos `ObjectCreator` que, haciendo uso de la reflexión de Java, analizará cada uno de los campos de la clase y asignará los valores correspondientes.

Para describir de forma más detallada la construcción aprovecharemos nuestro ejemplo de nuevo.
El método `createObject` de la clase recibirá la clase `Empleado` y el `ResultSet` obtenido de la ejecución.

Lo primero que necesitamos hacer es crear una nueva instancia del objeto con el método `newInstance` de la clase `Class`. Posteriormente, usando otro método de la misma clase (`getDeclaredFields`) obtendremos un `array` que contiene los campos privados de la clase en forma de instancias de la clase `Field`. En el caso de nuestro ejemplo los campos `nombre` y `edad`.

Recorremos este `array` mediante un bucle y, para cada uno de los campos que contiene, se obtendrá su nombre y se utilizará el método `getObject` de la clase `ResultSet` para recuperar su valor desde la base de datos.
Una vez obtenido el valor, se utilizará el método set de la clase `Field` para asignarlo al campo. Este método recibe el objeto al que se asignará el campo (nuestra instancia de la clase `empleado`) y el valor que hemos recuperado del `ResultSet`.

Cuando termina el bucle que recorre los campos tendremos en nuestra instancia el objeto construido que se devolverá en el método `createObject`.

El método `executeQuery` de la clase `Query` devuelve una lista con los objetos que cumplen las restricciones. En nuestro caso la lista contendrá únicamente el objeto que hemos recuperado durante este proceso pero si más de uno cumple las restricciones, se recorrerá el `ResultSet` utilizando el método `next` de la clase y, para cada uno de los resultados que contenga, se llamará al creador de objetos y se añadirán a la lista.

4.4.2. Query by example

Otra forma de recuperar objetos de la base de datos es mediante el tipo de consulta `QueryByExample` (consulta mediante ejemplo), que consiste en el uso de un *objeto* modelo cuyos atributos no nulos serán iguales a los del objeto que se quiere recuperar. La consulta se hará exclusivamente sobre los atributos del objeto modelo que sean distintos de cero o nulos.

Si el programador quisiera indicar en una consulta la restricción de que un determinado atributo tenga el valor `0` o `null` entonces puede indicar aquellos atributos que no quiere que se ignoren en la consulta.

Este modelo de consulta se ha implementado en el método `queryByExample` de la librería. Este método recibe el objeto modelo y la lista de campos que no se ignorarán.
Si el usuario de la librería no requiere el uso de la lista de campos a ignorar usará otro método del mismo nombre que recibe únicamente el objeto modelo, e internamente llamará al método nombrado anteriormente con una lista vacía.

Para implementar el proceso de consulta, `queryByExample` se apoya en las consultas descritas en el apartado anterior (3.4.1), de forma que, al recibir el objeto, mediante reflexión de Java comprobará cada uno de los campos y en caso de que

no sean cero, nulos o se encuentren en la lista de no ignorados se construirá la restricción de igualdad correspondiente.

Aplicándolo a nuestro ejemplo del `Empleado`, si queremos recuperar el objeto de la base de datos tendremos que seguir el siguiente proceso, que resulta más sencillo que la recuperación tradicional con restricciones:

1. Creación del objeto modelo con los mismos atributos que el que queremos recuperar (`nombre "Manuel"` y `edad 29`).
   ```
   Empleado modelo = new Empleado("Manuel",29);
   ```
2. Obtención de la lista de resultados y nuestro objeto recuperado mediante el método `queryByExample`.
   ```
   List<Empleado> l = (List<Empleado>) lib.queryByExample(modelo);
   Empleado manuel = l.get(0);
   ```

Ahora supongamos que la clase `Empleado` tiene un atributo más denominado `tiempo`, que representa los años que ha trabajado el empleado en la empresa. Nuestro empleado Manuel aún no ha llegado al año, por lo tanto su valor es 0.

Ahora queremos aplicar una nueva consulta sobre los empleados. Queremos saber todos los empleados que, como Manuel, aún no han superado su año trabajando. Si utilizamos el método `queryByExample` usando un modelo cuya variable tiempo valga 0 nos encontramos con el problema de que ese campo será ignorado, por lo que tenemos que hacer uso de la lista de campos no ignorados para que lo tenga en cuenta. De esta forma el proceso será el siguiente:

1. Creación del nuevo modelo. Los campos de nombre y edad se pondrán a nulo y 0 porque queremos ignorarlos.
   ```
   Empleado modelo = new Empleado (null,0,0);
   ```
2. Creación de la lista de los campos que no se ignorarán. En este caso añadimos el campo tiempo para que, a pesar de ser 0, no sea ignorado.
   ```
   List<String> atributos = new List<String>();
   atributos.add("tiempo");
   ```
3. Recuperación del objeto usando el método que incluye la lista y obtención del resultado (suponiendo que el único objeto que cumple la restricción del ejemplo en nuestra base de datos es Manuel).
   ```
   List<Empleado> l = (List<Empleado>) lib.queryByExample(modelo,atributos);
   Empleado manuel = l.get(0);
   ```

Durante este proceso el método `queryByExample` comprobará cada uno de los campos y creará la restricción en el caso de ser necesario.
En nuestro primer ejemplo la restricción final será una `AndConstraint` c compuesta de las `SimpleConstraint` c1 y c2 que se han construido de la siguiente forma:

c1: nombre del campo nombre, operando de igualdad (=) y valor del campo "Manuel"
c2: nombre del campo edad, operando de igualdad (=) y valor del campo 29

En nuestro último ejemplo la única restricción necesaria será de la forma `SimpleConstraint` con el nombre del campo tiempo, operando de igualdad y valor del campo 0.

En ambos casos tras la obtención de la restricción se asignará a una instancia de la clase `Query` creada automáticamente por el método `queryByExample` y se ejecutará de igual forma que las consultas tradicionales, devolviendo la lista de resultados al `queryByExample`, que será el encargado de devolverla al usuario.

Capítulo 5: Segunda fase: objetos con referencias simples

Una vez completada la primera fase en la que se ha implementado la inserción, actualización, borrado y carga de objetos con atributos de tipo básico, el siguiente paso a seguir ha sido la extensión de los procedimientos descritos anteriormente a objetos que contengan otros objetos como atributos, lo que se conoce en bases de datos relacionales como relación uno a uno y muchos a uno.

Al igual que en el capítulo anterior, durante el desarrollo de esta fase se recurrirá a otro ejemplo de objeto. Extenderemos nuestro ejemplo anterior con la inclusión de un nuevo tipo de objeto denominado `Direccion`, implementado como una clase. Esta clase se encuentra en los mismos paquetes que la clase `Empleado` del ejemplo del capítulo anterior (`ucm.tfg.Direccion`) y contiene dos variables de tipo simple: calle, de tipo `String`, y número, de tipo `int`. Hemos construido una instancia de esta clase a la que hemos llamado `facultad` con los valores `"Profesor José García Santesmases"` para calle y 9 para `numero`. Ahora ampliaremos nuestra clase de ejemplo `Empleado` añadiéndole un nuevo atributo de tipo `Direccion` llamado `direccion` y a la instancia que ya teníamos (empleado de `nombre` "`Manuel`" y `edad 29`) le asignaremos el objeto facultad descrito anteriormente como `direccion` (diagrama de clase: Figura 5.1).

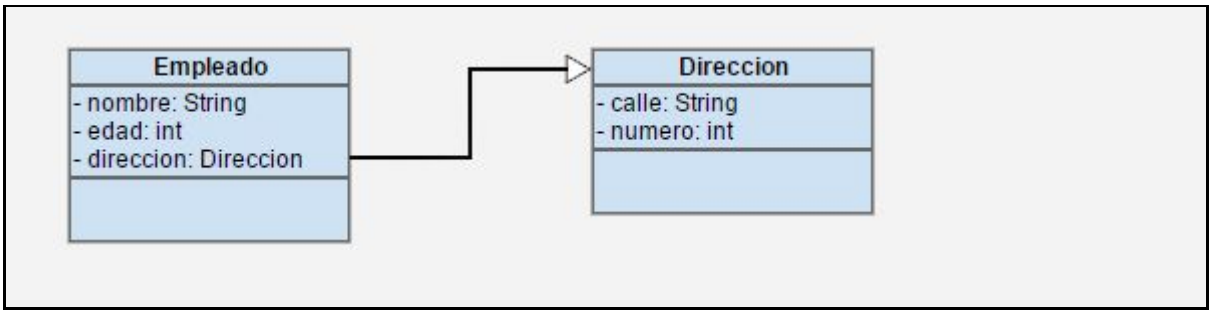

Figura 5.1

1. Referencias externas

1.1. Inserción

En esta segunda fase desechamos el método actualizar para hacer un único método `save`, con esto se trata de evitar el problema que ocurre cuando un objeto A apunta a un objeto B, estando B en la BD pero A no. Este método se encarga tanto de guardar objetos nuevos como de actualizar los objetos que ya se han guardado previamente, lo que en realidad

hace es actualizar siempre. Si el objeto ya estuviera se vería actualizado, si no estuviera se inserta una fila vacía y se actualiza también.
Esto lo hacemos así porque es necesario conocer el `ID` de la fila donde se va a insertar el objeto, y que otros objetos que le pudieran tener sepan el `ID` de su fila, así no hay problemas si se llama recursivamente a `save` con objetos que este pudiera tener.

Este método primero crea un mapa temporal de objetos visitados inicialmente vacío. Así se evita entrar en posibles ciclos (ver sección 5.2. el problema de los ciclos). Después llama a la clase `Saver` con el objeto a guardar. Esta clase es la encargada de ejecutar todo lo relacionado con las inserciones y las actualizaciones. Por último, actualiza el mapa de objetos guardados con el de objetos visitados.

El método `save` de la clase primero llamará a `createTable`. Este método creará una tabla para el objeto que se esté guardando. Esta tabla solo tendrá una columna `ID` única para cada objeto. Si esta tabla ya estuviera creada (esto se comprueba en el índice de tablas) no es necesario crearla. En cualquier caso devolverá el nombre de la tabla donde se guardan los objetos de esa clase.

Después se comprueba si el objeto ya ha sido guardado antes o visitado. En este caso se podrá obtener su `ID` de alguno de los dos mapas. En caso contrario, es necesario para la posterior inserción del objeto, modificar la tabla del objeto llamando a `alterTable`. Este método añadirá una columna para cada atributo básico para guardar su valor, y dos para los atributos no básicos una para guardar el nombre completo de la clase con la que se ha creado (ver sección 5.4.El problema de la herencia y las clases abstractas), y otra para guardar el `ID` del la fila donde se va a guardar ese objeto no básico.

Una vez que la tabla tenga suficientes columnas para guardar el objeto se llama a `insertEmptyRow`. Este método no insertará solo una fila vacía, ya que es necesario también insertar el nombre completo de la clase con el que se han creado las atributos no básicos (ver sección 5.4.El problema de la herencia y las clases abstractas), y se obtiene el `ID` de esta fila, que es donde se va a guardar el objeto.

Ahora teniendo el `ID` tanto de los objetos guardados antes o visitados y de los que se hace la primera inserción, se añadirá este objeto en el mapa de objetos visitados con su `ID` (ver sección 5.2. el problema de los ciclos), y en el mapa inverso.

Una vez hecho esto se procederá a recorrer los atributos del objeto y para cada atributo no básico que no esté en el mapa de visitados (ver sección 5.2. el problema de los ciclos) se llamara recursivamente a `save` ya que necesario guardar todos los objetos a los que apunta este primero antes de actualizar al final para conocer sus `IDs`, el propio método se encargará en la llamada recursiva de guardarlos y añadirlos al mapa con su `ID`.

Finalmente ya habiendo guardado todos los objetos que nuestro objeto pudiera tener, es posible llamar a `update`, este método recibe el objeto, el nombre de la tabla donde se va a guardar y el mapa de objetos visitados teniendo seguro que su `ID`, como de los objetos que este pudiera tener esta en el, con estos datos se construye una sentencia SQL que

actualizará para cada atributo básico su correspondiente columna con su valor, y para cada atributo no básico su correspondiente columna con el `ID` con que se ha guardado ese objeto en la base de datos.

1.2. Consultas

Tras la inclusión de los objetos como atributos de otros objetos, se ha introducido un nuevo concepto en la carga de objetos: la profundidad. La profundidad de un objeto X con respecto a Y es el número de punteros que hay que seguir desde Y hasta alcanzar a X. Un objeto está a profundidad 1 de sí mismo.
En el momento en el que ejecutemos nuestra consulta `Query` mediante el método `executeQuery` de la librería se tendrá que especificar la profundidad de carga que queremos que se alcance. En caso de no especificarse, se recurrirá a una variable global de la librería, que tiene un valor por defecto de 2, lo cual implica la carga de los objetos que cumplan las restricciones de la consulta y de los objetos a los que hagan referencia, pero si estos últimos tienen, a su vez, otras variables de tipo no básico, los objetos apuntados correspondientes no se cargarán y el valor de estas variables será nulo. Para cambiar el valor por defecto de la profundidad por defecto se ha incluido un método dentro de la librería.
La profundidad se tiene en cuenta a la hora de construir el objeto en la clase `ObjectCreator`. En el proceso de construcción, al comprobar que una clase es de tipo no básico, se comprobará también que la profundidad es mayor que cero. En el caso de serlo se ejecutará una sentencia que obtendrá el `ResultSet` del objeto correspondiente de la base de datos y se hará una llamada recursiva al método creador disminuyendo la profundidad en 1.

Aplicándolo a nuestro objeto de ejemplo, en el caso de que queramos cargar únicamente para operar con sus variables nombre y edad sin importarnos la dirección tendremos que ejecutar el siguiente código:

1. Creación de las restricciones y asignación al objeto `Query`.
   ```
   Query q = lib.newQuery(Empleado.class, lib);
   Constraint c1 = SimpleConstraint.newIgualQueConstraint(“nombre”, “Manuel”);
   Constraint c2 = SimpleConstraint.newIgualQueConstraint(“edad”, 29);
   Constraint c = new AndConstraint(c1,c2);
   q.setConstraint(c);
   ```

2. Ejecución del método `executeQuery`, especificando que el nivel de profundidad sea 1 y obtención del objeto.
   ```
   List<Empleado> l = (List<Empleado>) lib.executeQuery(q, 1);
   Empleado manuel = l.get(0);
   ```

Con este empleado cargado, si tratamos de ejecutar cualquier método de la clase `Direccion` con su variable `direccion` saltará una excepción del tipo NullPointerException, ya que, al haber recuperado el empleado con un valor de profundidad 1, el valor del atributo dirección es nulo.

Si, por el contrario, queremos cargar tanto el `Empleado` como su objeto interno `direccion`, podemos aprovechar la variable por defecto de la librería, no siendo necesario así especificar el nivel de profundidad al que queremos que se llegue. En este caso el proceso será exactamente igual que el anterior, salvo que en el punto 2 la primera línea se sustituirá por lib.executeQuery(q).

Además de incluir el concepto de profundidad a la hora de cargar un objeto, se ha introducido la posibilidad de aplicar restricciones sobre sus variables de tipo no básico. Para hacer esto, el nombre del campo sobre el cual se desea aplicar una restricción contendrá la secuencia de nombres de atributos, separados mediante puntos, que debemos seguir hasta alcanzar dicho campo.
Aplicando esto a nuestro ejemplo, la restricción de tipo simple que tendremos que crear sería la siguiente:
Constraint c = SimpleConstraint.newIgualQueConstraint("direccion.calle", "Profesor José García Santesmases").

Esta restricción se asignará a un objeto `Query` creado con la clase `Empleado`, así nos aseguraremos de que el objeto que se recupera es nuestro empleado, aunque la restricción haya sido aplicada sobre la `dirección`.

La recuperación a niveles de profundidad del objeto es independiente a la restricción aplicada sobre sus subobjetos. Es decir, aunque la restricción del ejemplo anterior se haya aplicado a la `dirección` podemos hacer una recuperación de nivel de profundidad 1 en la que la `dirección` tenga valor nulo.

Con esta nueva característica ha sido necesario cambiar la forma en la que se construyen las sentencias SQL de recuperación ya que ahora es necesario acceder a otras tablas externas a la de la clase sobre la que se aplica la consulta.
Anteriormente, cada restricción devolvía su correspondiente forma en lenguaje SQL. Esto ha sido necesario exteriorizarlo a la clase `Query`, que será la encargada de recorrer la restricción y, para cada una de las `SimpleConstraint` que la compongan, buscar las tablas correspondientes que, posteriormente, se incluirán en un `Join` de tablas de tipo `Left Join`.

Si aplicamos esto al ejemplo anterior, el proceso de construcción de la sentencia SQL (variable de tipo `String` llamada `sqlStatement`) será el siguiente:

1. Construcción del `SELECT` con los atributos de la tabla sobre la que se aplica la consulta (al ser la primera tabla el nombre asignado para las consultas ha sido t1). Los nombres de los atributos se han obtenido usando reflexión de Java.
   sqlStatement = "SELECT t1.nombre, t1.edad, t1.direccion";

2. Llamada a un método recursivo que obtendrá la forma SQL de las restricciones a aplicar. Este método recibe una lista de `String` sobre la que se añadirán cuando se llegue a un caso base (restricción `SimpleConstraint`) los nombres de las tablas que se deberán incluir en el `FROM`.

```
List<String> tablas = new ArrayList<String>();
String where = constraintToSql(this.constraint, tablas);
```

3. Construcción del `FROM` usando la lista de las tablas. Para cada lista se le asignará de forma ordenada un nombre de la forma `t + (índice de la lista + 2)`. Así aseguramos que la tabla con índice 0 sea la t2, índice 1 la t3, etc. Este nombre es el identificador SQL con el que se hará referencia a la tabla correspondiente en la consulta SQL.
   Las tablas se separan por `LEFT JOIN ON` y las variables que unen las tablas.
   sqlStatement += "FROM " + "1Empleado t1 LEFT JOIN 2Direccion t2 ON t1.direccion = t2.id";

4. Construcción del `WHERE` con la forma SQL de la restricción obtenida en el punto 2.
   sqlStatement += "WHERE " + where;

La sentencia resultado final de este proceso es la siguiente:

```
SELECT t1.nombre, t1.edad, t1.direccion
FROM 1Empleado t1 LEFT JOIN 2Direccion ON t1.direccion = t2.id
WHERE t2.calle = ?
```

Como se especificó en el capítulo anterior el signo de interrogación será sustituido por el valor que contenga la restricción.

Existe la posibilidad de que la restricción no se aplique sobre un campo de tipo básico como por ejemplo la `direccion` de la forma:

```
Constraint c = SimpleConstraint.newIgualQueConstraint("direccion",d1);
```

Siendo d1 una instancia de la clase `Direccion`.
En este caso al asignar el valor a la sentencia para sustituirlo por el signo de interrogación se buscará en el mapa de identidad su `ID` de la tabla que le corresponde. De esta forma aseguramos que el objeto que pasamos como valor sea sobre el que realmente queremos comparar y no otro que tenga sus mismos valores.
Teniendo esto en cuenta, si hacemos el siguiente proceso:

```
Query q = lib.newQuery(Empleado.class,lib);
Direccion d = new Direccion("Profesor José García Santesmases",9");
Constraint c = SimpleConstraint.newIgualQueConstraint("direccion",d);
q.setConstraint(c);
List<Empleado> lista = (List<Empleado>) lib.executeQuery(q);
```

La lista de resultados lista estará vacía ya que, a pesar de que en nuestra base de datos existe un `Empleado` con una `direccion` cuyos valores son iguales que los de la `Direccion d` creada en el proceso, no se tratan del mismo objeto y la restricción no se aplica sobre él.

2. El problema de los ciclos

Tras incluir la posibilidad de almacenar variables de tipo no básico surge un nuevo problema en la carga de objetos: los ciclos, o referencias mutuas entre distintos objetos. Cuando

tratamos de cargar un objeto cuyo árbol de variables, incluyendo todos los niveles, contiene un ciclo cerrado surgen problemas a la hora de crear dicho objeto.
Para solucionar este problema se ha decidido usar otro `IdentityHashMap` llamado `im`. Aquí guardaremos como clave un `Object` y como valor el id de su fila en la tabla de la base de datos. Este `IdentityHashMap` se inicializará en cada llamada al método `save` ya que `im` lo utilizaremos para ir metiendo los objetos que vayamos guardando en la base de datos, para que si un objeto ya está en `im` eso quiere decir que ya ha sido guardado y no se deberá guardar de nuevo para que no se produzca un ciclo.

Poniendo como ejemplo nuestro `Empleado`, aunque haciendo una modificación en la `Direccion` (diagrama de clase: Figura 5.2) explicaremos el problema de los ciclos. Ahora la `Direccion` contendrá un atributo más, que será un `Empleado`, el cual será Manuel.

Figura 5.2

1. Lo primero que haremos será llamar al método guardar pasandole nuestro `Empleado`.
   ```
   libreria.save(empleado);
   ```
2. Esto lo que hará será llamar a un método auxiliar donde se creará el `IdentityHashMap` y se llamará al `save` de la clase `Saver`.
   ```
   IdentityHashMap<Object, Integer> im=new IdentityHashMap<Object, Integer>();
   this.saver.save(objeto, im);
   ```
3. Crearemos la tabla de empleado si no está creada ya.
4. Si `im` ni `objectMap` contiene a nuestro empleado Manuel, alteramos la tabla metiendo las columnas correspondientes e insertándolas en el `indicecolumna` en la base de datos e insertamos una fila vacía la cual nos devolverá su `ID`. En caso de que esté en alguno de los dos mapas cogeremos el `ID` del mapa.
5. Después meteremos en `im` a nuestro empleado Manuel.
   ```
   im.put(objeto, id);
   ```
6. Recorreremos los atributos de empleado y, en caso de que no sea básico, habremos llegado a `Direccion`. Miraremos si está contenido en `im`. Si está no deberemos guardarlo ya que habrá sido guardado anteriormente. Si no está llamaremos de nuevo a `save`, pero ahora con `Direccion`.
7. Volveremos hacer todos los pasos a partir del paso 3. Ahora cuando recorramos los atributos de `Direccion` y lleguemos a `Empleado`, no lo guardaremos de nuevo en la BD, ya que estará contenido en `im` y, por lo tanto, no se formará ningún ciclo.

3. Activación por niveles

Como ya se ha descrito en el apartado 1.4 de este mismo capítulo, se ha incluido el concepto de profundidad a la hora de cargar un objeto. Una vez cargado, todos los atributos que queden por encima del nivel de profundidad elegido serán nulos. No obstante, puede ocurrir que posteriormente el usuario de la librería desee acceder a los valores. Por lo tanto para complementar la carga de objetos hemos implementado el método de activación.

Con este método lo que queremos conseguir es que, dados un objeto y su profundidad (por defecto 2), recupere el objeto a la profundidad indicada. En nuestro ejemplo si pasamos a este método nuestra instancia de `Empleado` y un valor de 1, no nos recuperara la `Direccion` ya que esta se encuentra a nivel 2 de profundidad.

Para la activación de objetos debemos comprobar que el objeto sobre el que se aplicará la activación esté contenido en el `objectMap` y por lo tanto, se encuentre en la base de datos. Esto es debido a que, en caso contrario, no habría forma de conocer su `ID`. El método de activación será recursivo, donde el caso base corresponde al valor de profundidad cero, y las llamadas recursivas decrementan en uno el valor de profundidad.

Una vez que no estemos en nuestro caso base, recorreremos los atributos del objeto y comprobaremos, para cada uno de ellos, que no es de tipo simple (si lo fuera no haría falta activarlo, ya que no hay ningún nivel por debajo de un objeto simple), y que su valor es nulo. En este caso de que haya un atributo que cumpla estas condiciones, recuperamos su valor de la BD, pero restando 1 a la profundidad.

Pasada esta condición tendremos que saber el id de la fila de del objeto padre y el nombre de su tabla en la base de datos para recuperar todos los datos de esa fila y quedarnos con el id que está contenido en el nombre de la columna de nuestro objeto inicial y además rescatar el nombre de la tabla en la base de datos. De la misma forma que hemos descrito anteriormente, recuperamos toda la fila de la base de datos y, pasándola al método `createObject` junto con la profundidad, nos devuelva el objeto recuperado, que asignaremos al atributo correspondiente, cuyo valor anterior era nulo.

Poniendo como ejemplo nuestro `Empleado` y suponiendo que se ha cargado con profundidad 1, realizaremos una activación con profundidad 2.

1. El programador llamará al método activar.

   ```
   libreria.activate(empleado, 2);
   ```

2. Como nuestra profundidad no es 0, procederemos a recorrer los atributos del objeto `Empleado`. Una vez lleguemos a `Direccion`, conseguiremos su valor (que será nulo), por lo que recuperaremos el objeto reemplazando el valor nulo por el objeto recuperado. En `field.getType()` pasaremos la clase de nuestra `Dirección` de la cual podremos sacar su ruta en el proyecto para sacar el nombre de la tabla en la base de datos.

   ```
   field.set(empleado, recuperar(empleado, field.getName(), field.getType(), profundidad-1) );
   ```

3. Ahora la profundidad con la que se recuperará la `Direccion` del Empleado será 1, así que tendremos que conseguir el id del `Empleado` (será el `ID` 1) y el nombre de su tabla la cual será `1Empleado`. Con estos dos datos podremos realizar la consulta sql para que nos devuelva su fila:

   Select * from 1Empleado where id = 1

4. De esta fila recuperada, lo único que deberemos coger es el `ID` contenido en la columna con el nombre `direccion`. Este será el `ID` de la fila en la base de datos (en la tabla de del objeto `Direccion`) que queremos recuperar. Este `ID` será 4. Tendremos que buscar también el nombre de la tabla de la base de datos en el `indiceTabla`. Teniendo estos datos podremos realizar la consulta SQL para que nos devuelva la fila de la `Direccion` que queremos rescatar.

   Select * from Direccion2 where id = 4

5. Esta fila será pasada al método `createObject` junto con la profundidad, que en este momento es 1, y un objeto `Class` que será la `Direccion`. Esto nos devolverá un objeto `Direccion` ya creado.

   objeto = query.createObject(classDirecion, resultSetDireccion, profundidad);

6. Este objeto será devuelto al método principal y el atributo correspondiente será actualizado de la forma explicada anteriormente..

4. El problema de la herencia y clases abstractas

Junto a las novedades implementadas descritas en este capítulo ha surgido un problema durante el proceso de construcción de un objeto tras la carga: la creación de variables que se han declarado con una clase de tipo abstracto.

El proceso de construcción de un objeto que intentamos cargar se apoya en la reflexión de Java para obtener el nombre y tipo de cada uno de sus campos, pero en el caso de que un campo esté declarado como instancia de una clase abstracta, nos es imposible conocer el tipo que hereda de dicha clase con el que se construyó el objeto realmente.

Para solucionar este problema hemos incluido una nueva columna en la tabla correspondiente a cada objeto en la base de datos. En esta columna se insertará el nombre cualificado de la clase, incluyendo paquetes, con la que se ha construido el objeto que queremos guardar. El nombre de esta columna estará compuesto por el número 2 seguido de un guión bajo seguido del nombre del campo. Este nombre se ha decidido teniendo en cuenta que las variables en Java no pueden empezar por número y así no entrar en conflicto con el nombre de las columnas de otros atributos.

Tras esto, a la hora de construir el objeto, en vez de obtener la clase de cada campo mediante el método `getType` de la clase `Field`, se usará el método `fromName` de la clase `Class`, que permite obtener una nueva instancia de una clase a partir de su nombre completo.

Para ilustrar de forma más clara este proceso haremos uso de nuestro ejemplo aplicando algunos cambios. Nuestra clase `Direccion` ahora es una clase abstracta de la que

cuelgan dos clases que la heredan, la clase `Calle` y la clase `Plaza` (diagramas de clase: Figura 5.3).
El diagrama de clases de la clase `Empleado` no sufre cambios pero en nuestra instancia la variable `direccion` ahora está creada como instancia de la clase `Calle`.

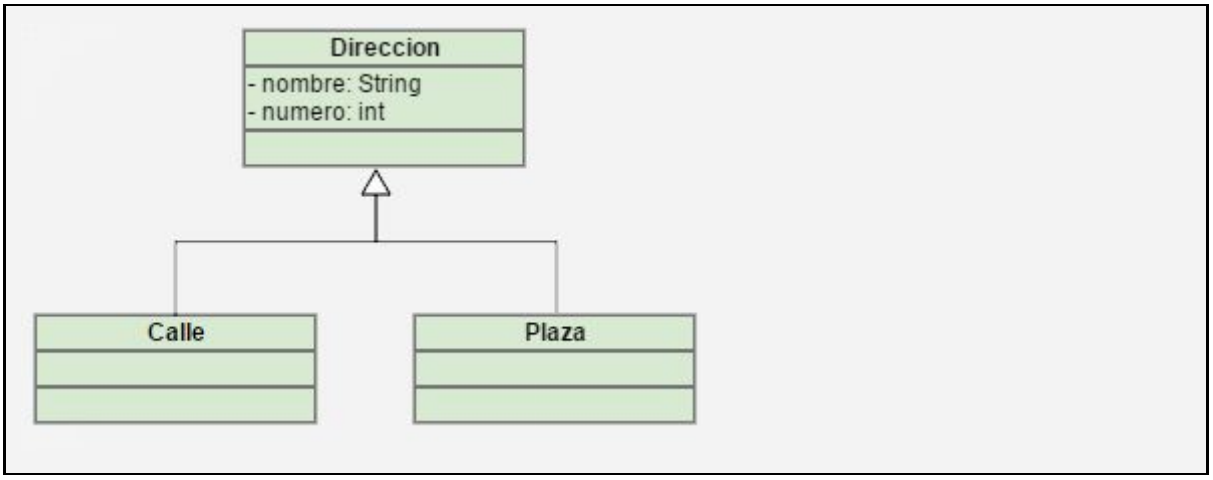

Figura 5.3

Si no tenemos en cuenta la resolución del problema y tratamos de cargar nuestro `Empleado`, a la hora de construirlo nos encontramos con que la clase a la que pertenece la variable direccion es `Direccion`, pero no sabemos si en realidad es `Calle` o `Plaza`.

Con la incorporación del nuevo atributo a la tabla de empleados, si guardamos el `Empleado` en la base de datos, quedará una fila con la siguiente información:

ID	**Nombre**	**Edad**	**Direccion**	**2_direccion**
1	Manuel	29	1	ucm.tfg.Calle

Teniendo esta información almacenada, en el momento en el que queramos construir el campo `direccion` tendremos que usar la llamada `Class.forName("ucm.tfg.Calle")`, que nos devolverá una nueva instancia de la clase `Calle` lista, para asignar los valores correspondientes a sus atributos.

Capítulo 6: Tercera fase: objetos con referencias múltiples

En esta tercera y última fase del proyecto se implementará la inserción, actualización y carga de variables de tipo multivalorado pertenecientes a otros objetos.

Como ya se hizo en el capítulo anterior, extenderemos, de nuevo, nuestro ejemplo para adaptarlo a las nuevas implementaciones incluidas en este capítulo.
La clase `Direccion` se mantendrá igual que en el capítulo anterior, pero en la clase `Empleado` cambiará el tipo de referencia. En este capítulo la variable `direccion` de la clase `Empleado` pasará a llamarse `direcciones` y cambiará su tipo de `Direccion` a `List<Direccion>`. Además se incluirá una nueva variable de tipo `Set<Integer>` denominada `telefonos`.
El diagrama de clases del ejemplo descrito en el párrafo anterior puede verse en la figura 6.1.

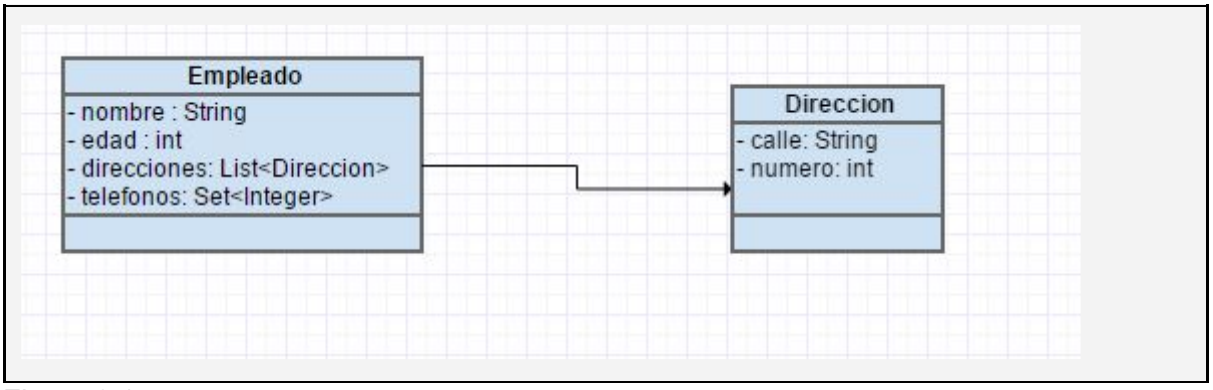

Figura 6.1

Teniendo esto en cuenta, creamos dos objetos de tipo `Direccion`, a los que hemos llamado `casa` y `sede`, que representan el domicilio personal del empleado y la dirección de la sede de la empresa en la que trabaja respectivamente. La `Direccion casa` se ha creado con `calle "Embajadores"` y `numero 23` y la `Direccion sede` con `calle "Preciados"` y `numero 12`.
Ambas direcciones se han añadido a la `List<Direccion> direcciones` del `Empleado` de ejemplo Manuel.
Al `Set<Integer> telefonos` se han añadido los números `686465387` y `914563765`.

1. Tipos de referencias múltiples

En esta fase, donde se permiten relaciones múltiples, hay varias formas de representar esta relación, pero para todas ellas es necesario guardar las múltiples referencias que un objeto puede tener de otro. Para esto es necesario crear una tabla para cada relación múltiple.

Por otra parte, en este capítulo también se tienen en cuenta los atributos multivalorados. Este tipo de atributos, al igual que en las relaciones varios a varios, contienen varios valores. Sin embargo, los atributos multivalorados no hacen referencia a otras entidades de la BD (objetos, en el caso de nuestra librería), sino que contienen una colección de valores básicos.

`Set`: para sus dos implementaciones (`HashSet` y `TreeSet`) es necesario crear una tabla con tres columnas: una para el `ID` que hará de clave de la tabla, una columna para el `ID` del objeto contenedor de la referencia y otra columna para el `ID` de los objetos a los que hace referencia.
`List`: para sus dos implementaciones (`ArrayList` y `LinkedList`) es necesario crear una tabla que, aparte de las columnas anteriores, tenga una columna con la posición de cada elemento, ya que existe un orden determinado entre los elementos de la lista.
`Array`: La tabla creada aquí es muy parecida a la de las listas. También existe una posición, pero aquí el tamaño del `array` es fijo, pudiendo estar algunas posiciones vacías.

Además, si estas tres implementaciones se usan con objetos básicos y no con referencias a otros objetos solo es necesario guardar el valor del objeto básico en lugar de la referencia al mismo.

2. Cambios en los métodos

2.1. Inserción

Al incluir los tipos de referencias múltiples se han tenido que realizar algunas modificaciones en el código. Cuando guardamos un atributo del objeto que no es básico ahora, además, deberemos mirar si este atributo implementa `List`, `Set` o es de tipo `array`. En caso de que sea uno de estos tres tipos se necesitará saber qué tipo de parámetro tienen. Aquí nos encontraremos dos tipos:

- Si el parámetro es básico, este atributo funciona como un multivalorado en SQL. Esto implica que crearemos una tabla en la base de datos. El nombre de esta tabla se generará juntando el nombre de la tabla de la clase en la que está el atributo más una barra baja más el nombre del atributo. Esta tabla se creará en el método `getAttributes`, donde se mirara si el atributo es `List`, `Set` o `Array`.

- Si el parámetro no es básico crearemos una tabla intermedia en la base de datos, cuyo nombre se obtendrá juntando el nombre de la tabla de la clase en la que está el atributo más una barra baja más el nombre del atributo. Antes de insertar una fila en esta tabla intermedia, deberemos guardar el objeto al que esta fila hará referencia, ya que necesitaremos su `ID` en la tabla de la base de datos en la que se guarde este objeto. Esta tabla se creará en el método `getAttributes` donde se mirara si el atributo es `List`, `Set` o `Array`.

Poniendo como ejemplo el explicado en la figura 6.1.

1. Al recorrer la lista de atributos en el método `getAttributes`, llegaremos al atributo direcciones. Al ser una instancia de `List` comprobaremos si el parámetro del tipo (`Dirección`) es de tipo básico. Como no lo es, tendremos que crear una tabla intermedia llamada `1empleado_direcciones` con las columnas `id`, `id1_1Empleado`, `id2_2Direccion` y `posicion`, como podemos ver en la figura 6.2

Figura 6.2

2. Al recorrer el atributo `telefonos`, vemos que es una instancia de `Set` y que el parámetro del tipo es `Integer`, por lo que se tendrá que crear una tabla con las columnas `id1_1Empleado` y `telefonos`, como podemos ver en la figura 6.2

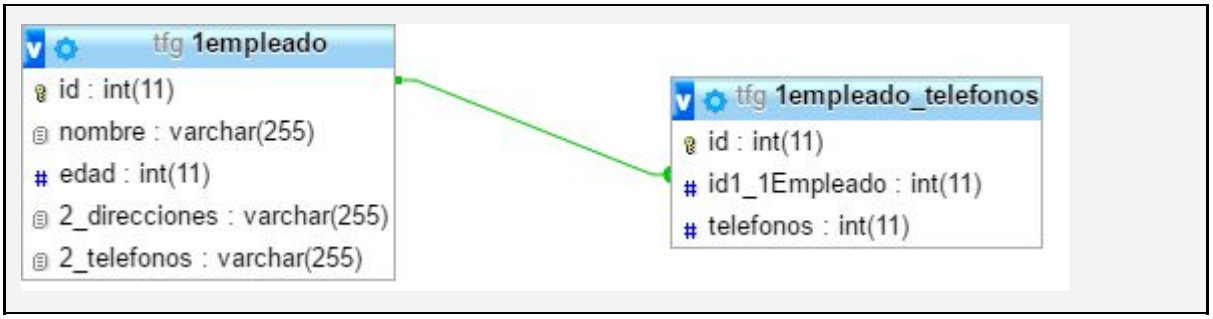

Figura 6.3

3. Recorremos de nuevo la lista de atributos para realizar las inserciones de los elementos relacionados. Una vez que lleguemos al atributo `direcciones` al ser un `List` de `Direccion` tendremos que recorrer las dos `direcciones`:
    a. Guardaremos la primera `direccion` en su tabla correspondiente y posteriormente insertaremos una fila en la tabla intermedia. Suponiendo que el ID de nuestro `empleado` es 1 y el ID de la `direccion` es 2, en la tabla intermedia guardaremos un 1 en la columna `id1_1Empleado`, un 2 en la columna `id2_2Direccion` y un 1 en la `posición`.

ID	id1_1Empleado	id2_2Direccion	posicion
1	1	2	1

b. Guardaremos la segunda `direccion` en su tabla correspondiente y posteriormente insertaremos su fila en la tabla intermedia. El `ID` de nuestro empleado volverá a ser 1 y el `ID` de la nueva `direccion` es 3. En la tabla intermedia guardaremos un 1 en la columna `id_1Empleado`, un 3 en la columna `id2_2Direccion` y un 2 en la `posición`.

ID	id1_1Empleado	id2_2Direccion	posicion
1	1	2	1
2	1	3	2

4. Cuando lleguemos al atributo `telefonos`, al ser un `Set` de `Integer`, tendremos que recorrer los dos `telefonos`:
   a. Con el primer teléfono, al ser de un tipo básico solo lo tendremos que guardar en la tabla intermedia correspondiente a este atributo multivalorado. Supongamos que el `ID` de nuestro empleado es 1. En la tabla de multivalorado guardaremos un 1 en la columna `id1_1Empleado` y `686465387` en la columna `telefonos`. Al ser `Set` no guardará ninguna posición del teléfono dentro del `Set`.

ID	id1_1Empleado	telefonos
1	1	686465387

b. Con el segundo `telefono` , suponiendo que el `ID` de nuestro empleado es 1, en la tabla de este atributo guardaremos un 1 en la columna `id1_1Empleado` y `914563765` en la columna `telefonos`, de nuevo, sin incluir su posición.

ID	id1_1Empleado	telefonos
1	1	686465387
1	1	914563765

2.2. Actualización

Al incluir los objetos con referencias múltiples no realizamos una actualización como tal sino lo que hacemos es vaciar la tabla intermedia o la tabla de multivalorado según el caso en el que nos encontremos del objeto que estamos intentando actualizar. Aunque esto se realiza en el método `save`.

Para vaciar la tabla como las dos ya sea multivalorado o intermedia se crean con el mismo nombre, es decir, nombre de la tabla del objeto en la base de datos mas guion bajo más nombre del atributo, y las dos tienen una columna que se llama “id1_nombre de la tabla en la base de datos”.
Lo único que necesitaremos es pasarle estos datos al método que vacía la tabla. Este realizara una consulta a la base de datos borrando las filas donde coincida el `ID` de nuestro objeto con la columna anteriormente nombrada.

2.3. Borrado

El borrado se sigue realizando como la primera fase. Se pasa un objeto y solo se borra la fila en la tabla de la base de datos de ese objeto.

2.4. Consultas

Al incluir los atributos de tipo `List`, `Set` y `Array` ha sido necesario modificar el constructor de objetos para soportar la creación de objetos de estos tres tipos.
En el método `createObject` se han incluido tres nuevas condiciones para comprobar si el campo que estamos analizando es de los tipos mencionados en el párrafo anterior.

En el caso de que sea de tipo `List` o `Set` hay que tener en cuenta que estos dos tipos corresponden a clases abstractas. Por tanto se hará una comprobación adicional consultando en el `ResultSet` el tipo con el que fueron construidas en el objeto guardado. Para `List` se diferencia entre `ArrayList` y `LinkedList`. En el caso de `Set` se diferencia entre `HashSet`, `LinkedHashSet` y `TreeSet`.

Una vez conocido el tipo de `List` o `Set` con el que fue construido el campo creamos una instancia de dicho tipo, que será la que posteriormente sea asignada al atributo correspondiente del objeto que estamos creando. Tras esta creación comprobamos el tipo de objeto parametrizado en la `List` o `Set`, para poder diferenciar su tipo entre básico o no básico.

En el caso de que se trate de un tipo básico se hará una consulta a la tabla correspondiente al campo en la que encontraremos toda la información que necesitamos.
Si, por el contrario, el objeto no es de tipo básico será necesario hacer una consulta a la tabla del campo para obtener los `ID` de los objetos guardados en la `List` o `Set` y posteriormente una consulta adicional por cada `ID` para recuperar el objeto y construirlo a través del `ResultSet` obtenido y el método `createObject`.

En el caso de que el campo no sea de tipo `List` o `Set`, sino `Array`, el proceso de construcción será algo diferente.
Mediante la clase `Array` podemos utilizar el método `newInstance` para construir el campo como un `Array`. Este método recibe la clase de los componentes del `Array`, que obtendremos mediante el método `getComponentType()` de la clase Class asociada al campo `Field` que estamos construyendo y la longitud del `Array`, que obtendremos de la base de datos.
Una vez tenemos el campo instanciado como un `Array` tan solo tenemos que hacer uso del método `set` de esta última clase, que recibe el objeto `Array` campo, la posición del elemento en el `Array` y el valor que dicho elemento debe tomar.

Además también se ha incluido la posibilidad de hacer consultas sobre este tipo de variables.
Para hacer consultas sobre si un objeto contiene o no un objeto dentro de `List`, `Set` o `Array` se ha hecho uso de la `SimpleConstraint` de igualdad (=).

Aplicando esto a nuestro ejemplo, si queremos consultar si la variable `telefonos` tiene el número `914563765` tendremos que crear la siguiente `Constraint`:
```
Constraint c = SimpleConstraint.newEquals("telefonos",914563765);
```

Para implementar la cláusula `WHERE` de la sentencia `SQL` a ejecutar hay dos alternativas: utilizando el operador `IN` o utilizando el operador `EXISTS`.
Se ha decidido implementarlo usando el operador `EXISTS`, ya que hace más fácil la inclusión del `NOT` en caso de serlo (`ID NOT IN` frente a `NOT EXISTS`).

Si asignamos la restricción de ejemplo anterior (`c`) a una `Query` y ejecutamos la construcción de la condición del `WHERE`, utilizando una variable de ejemplo llamada `whereStatement` correspondiente a la `Constraint c`, se hará de la siguiente forma:

1. Se comienza a construir la cláusula añadiendo el `EXIST` y el comienzo de la sentencia interna. En esta sentencia se identifica la tabla correspondiente a la lista como `ntl`.
   ```
   whereStatement = "EXISTS (SELECT ntl.id1_1Empleado";
   ```

2. Se añade la parte correspondiente al `FROM`.
   ```
   whereStatement += "FROM 1Empleado_direcciones ntl";
   ```

3. Se añade la parte del WHERE con las condiciones necesarias.
   ```
   whereStatement += "WHERE ntl.felefonos = ?
        AND ntl.id1_1Empleado = t1.id";
   ```

También podemos consultar sobre la lista `List<Direccion>` para comprobar que el Empleado de ejemplo contiene una `Direccion` determinada. Hay que tener en cuenta que la comprobación es sobre la identidad del objeto, y no sobre su contenido. Es decir, si creamos una `Direccion` con los mismos valores que una ya existente en la BD la incluimos en la `SimpleConstraint`, no se recuperará el objeto. Se tendría que usar los

objetos ya creados o recuperados de la base de datos durante la sesión de uso de la librería.

Teniendo esto en cuenta, creamos la siguiente restricción:

```
Constraint c = SimpleConstraint.newEquals("direcciones",casa);
```

En este caso el proceso de construcción de la cláusula `WHERE` será similar pero en este caso en la parte correspondiente al `FROM` es necesario hacer un `JOIN` con la tabla correspondiente al tipo de objeto ya que contiene información necesaria para la restricción.

Capítulo 7: Conclusiones y trabajo futuro

1. Objetivos cumplidos

1.1. Diseño de API para librería BD orientada a objetos

En la primera fase de desarrollo se ha cumplido uno de los objetivos que teníamos, empezando por el diseño e implementación de una API de una librería de bases de datos orientada a objetos con atributos simples. La librería ha hecho uso de una base de datos de tipo relacional.

Se pueden realizar gestiones como inserción, actualización, borrado o carga sin necesidad de que el usuario construya las sentencias SQL.

Se dispone de un sistema de consultas con restricciones con el cual se podrá recuperar objetos que solo constan de atributos de tipo básico en Java.

Además del método anterior de consultas dispone de una forma de especificar consultas llamada `queryByExample`, que recibe un objeto prototipo y devolverá todos los objetos que tengan los mismos valores que dicho prototipo en sus atributos no nulos.

Además de los dos métodos de consulta nombrados anteriormente, se tenía planeado otro modo adicional usando predicados Java, pero por el abandono de un integrante del grupo no se pudo continuar con la idea y se desechó por falta de tiempo.

1.2. Extensión para soportar relaciones entre objetos

Al igual que el objetivo anterior, se han cumplido todas las expectativas de este objetivo.

La librería acepta la inserción, actualización y borrado de objetos con relaciones con otros de tipo no básico y permite la recuperación de dichos objetos aplicando restricciones sobre sus relaciones.

Durante la implementación de este objetivo han surgido dos problemas que se han resuelto con éxito: la existencia de ciclos generados por las relaciones entre objetos y la identificación de la clase cuando se trata de almacenar objetos pertenecientes a una clase abstracta.

1.3. Extensión para soportar relaciones de multiplicidad

En esta fase se ha cumplido el objetivo de implementar la inserción, actualización y borrado de un objeto con relaciones de multiplicidad al completo.

La librería acepta la recuperación de objetos aplicando restricciones sobre el contenido de los atributos de tipo múltiple pero no se ha llegado a implementar la posibilidad de aplicar restricciones sobre los atributos de los objetos que se encuentran dentro de los `List`, `Set` o tipo `array`.

Globalmente podemos afirmar que sí hemos cumplido los objetivos propuestos en gran medida y como resultado hemos obtenido una librería que puede utilizarse para el manejo de las bases de datos de aplicaciones. Hemos adquirido conocimientos más allá de los aprendidos en asignaturas similares a este proyecto como es Java Reflection y GitHub, además de perfeccionar conceptos dados en clase.

2. Trabajo futuro

Si continuáramos trabajando en este proyecto implementaríamos las siguientes características en la librería:

- Extensión de las restricciones para que se permita la posibilidad de aplicar restricciones sobre los objetos que están contenidos dentro de atributos de tipo múltiple pertenecientes a otros objetos.

- Mejora del método de guardado para facilitar el uso al usuario final. El método de guardado actual de la librería admite únicamente un objeto. Como trabajo futuro se puede extender el método para admitir más de un objeto o una lista de objetos para guardarlos al mismo tiempo, en vez de tener que realizar una llamada por cada objeto que se desea guardar.

- Extensión para soportar atributos de tipo tabla asociativa (`Map`) con clave y valor. Al igual que con los atributos de tipo `List`, `Set` o tipo `array` se diseñaría la tabla resultante en la base de datos y se implementaría la inserción, actualización, borrado y consulta de este tipo de objetos.

- Añadir la posibilidad de generar índices sobre campos de los objetos. Esto se realizaría incorporando dos métodos: uno que recibirá el objeto y todos los nombres de los atributos a los que añadir un índice y otro que recibira también el objeto y todos los nombres de los atributos, pero añadirá un único índice para todas esas columnas.

- Extender el método save para que en vez de que actualice un objeto guarde otra versión de ese mismo objeto, será necesario aparte del id añadir una columna con la fecha a la que se guardó ese objeto, para así diferenciar las distintas versiones del mismo objeto, y a la hora de recuperar, que recuper siempre la última por defecto, o usando otro método que te devuelva las distintas versiones del mismo.

- Implementación de un mecanismo similar a las consultas nativas de db4o. Mediante este tipo de consultas el programador puede expresar las condiciones de búsqueda

mediante un método Java que, dado un objeto, devuelva un booleano indicando si cumple dichas condiciones de búsqueda. Para realizar este tipo de búsquedas sería necesario un análisis del *bytecode* correspondiente por el método booleano para extraer, en casos sencillos, una representación SQL de la condición correspondiente.

- Método para crear la base de datos en MySQL, de modo que el usuario no tenga que crearla manualmente mediante phpMyAdmin a la hora de instalar el sistema. A partir del método, ya existente, que recibe el nombre de la base de datos, nombre de usuario y contraseña, se añadirá la funcionalidad de creación de la base de datos en caso de que no exista, creación de usuarios en el SGBD, y conexión a la base de datos.

Capítulo 8: Conclusions and future work

1. Accomplished objectives

1.1. Object oriented DB library API design

In the first phase of the development the objectives have been accomplished, starting from the design and implementation of an object oriented database library API with basic attributes. The library uses a relational database.

Database operations such as insertion, update, deletion, and retrieval can be done without the user needing to construct the SQL statements himself.

The library has a query system with constraints you can use to retrieve objects with basic type attributes.

Also, the library includes another query system besides the one described previously named `queryByExample`, that receives a prototype object and returns all the objects that match its attributes except null.

Another query system using Java predicates was going to be implemented but it had to be abandoned due to a member of the group abandoning the project and it was rejected due to the lack of time.

1.2. Extension to support relations between objects

Same as the previous objective, every expectation of this one has been accomplished.

The library accepts the insertion, update and deletion of objects with non-basic type object relations and allows the retrieval of these objects by applying constraints over its relation.

During the implementation of this objective we have encountered and solved two problems: the existence of cycles generated by the relations between the objects and the identification of the class when it comes to objects belonging to an abstract class.

1.3. Extension to support multiplicity relations

In this phase we have completely accomplished the objective of implementing the insertion, update and deletion of an object with multiplicity relations.

The library accepts the retrieval of objects by applying constraints over the multivalued attributes but the possibility of applying constraints over the attributes of the objects inside the `List`, `Set` or `array` type objects has not been implemented.

Globally we can say that we have accomplished the proposed objectives and as a result we have got a library that can be used to manage applications databases. We have acquired knowledge beyond the ones we had already obtained in similar subjects to this project such as Java Reflection and GitHub, besides improving the studied ones.

2. Future work

If we continued working in this project we would implement the following features in the library:

- Extension of the constraint system so it allows the possibility of applying constraints over objects inside multivalued attributes.

- Improving the save method to make it easier for the final user. The current method for saving admits only one object. We could extend the method so it admits more that one or a list of objects to save them all at once instead of calling the method for each one.

- Extension to support `Map` type attributes. Same as the `List`, `Set` or `array` type objects we would design the resulting table in the database and the insertion, update, deletion and load would be implemented.

- Add the possibility of generating index over the fields of an object. We would incorporate two methods: one that would receive the object and every name of the attributes the user wants to add an index over and other method that would receive the object and every name of the attributes, but would only generate one index for all the columns.

- Extend the save method so that instead of updating a given object, to save another version of the same object.In order to achieve this it would be necessary to add another column with the date the object was save, so we can differentiate between the different versions of the same object and when it comes to retrieving it, retrieve the last version as default or selecting the wanted version using another method.

- Implementation of a native query system similar to db4o. With this kind of query system the programmer can state the search conditions using a Java method that, given an object, returns a boolean indicating if it satisfies said search conditions. To implement this kind of system it would be necessary to do an analysis of the bytecode of the boolean method to get, in simple cases, a SQL representation of the constraint.

- Method for creating the MySQL database so the user does not have to create it manually using phpMyAdmin when he is setting up the system. Using the existing method that receives the name of the database, the user name and the password it

would be implemented the feature of creating the database if it does not exist, along with the user in the DBMS and the connection to the database.

Apéndice 1: API de la librería

- `OODBLibrary(String dbName, String user, String pass)`

Constructor de la librería, recibe en nombre de la base de datos a la que conectarse, el nombre del usuario que se va a usar y su contraseña. Devuelve la librería ya inicializada y conectada.

- `void save(Object o)`

Método para guardar un objeto en la librería. Este método guardará el objeto y todos los objetos a los que referencie. Si el objeto ya estaba guardado anteriormente, actualizará sus datos.

- `void delete(Object o)`

Recibe el objeto que será eliminado de la base de datos. Este objeto debe haber sido guardado o recuperado previamente. Si no terminara con excepción `NonExistentObject`. Si el objeto recibido tiene referencias a otros objetos, estos últimos no serán eliminados.

- `List<Object> queryByExample(Object o)`

Recibe un objeto que usa como prototipo para devolver todos los objetos del mismo tipo que tengan los mismos valores en sus atributos, ignorando los atributos que tengan el valor `null` o `0`.

- `List<Object> queryByExample(Object o, List<String> notToIgnore)`

Este método se usa para devolver todos los objetos que que tengan los mismos valores en sus atributos que el objeto recibido, pero no ignorará los atributos con valor nulo o cero de los atributos que estén en la lista recibida.

- `Query newQuery(Class<?> class)`

Método para crear consultas. Recibe la clase del objeto sobre el qué hacer consultas y devuelve un objeto que representa la consulta que por defecto devuelve todos los objetos de esa clase que haya en la base de datos. Es posible añadir una condición de búsqueda usando su método: `void setConstraint(Constraint c)`

- `List<Object> executeQuery(Query q)`

Este método ejecutara la consulta que recibe y devuelve todos los objetos que cumplan la restricción contenida dentro del objeto q. Se devolverá cada uno de los objetos que cumplan la restricción, incluyendo aquellos objetos referenciados por estos, pero solo hasta una profundidad máxima por defecto.

- `void setDepth(int depth)`

Como algunos de los siguientes métodos usan una profundidad máxima por defecto, con este método es posible modificar dicha profundidad.

- `List<Object> executeQuery(Query q, int depth)`

Recibe, además de la consulta a ejecutar, un nivel de profundidad, que será el que se usará para devolver los objetos a los que referencie, a partir de aquellos que cumplen la restricción de la consulta.

- `void activate(Object o)`

Dado que al recuperar un objetos solo recupera sus referencias hasta una cierta profundidad, es necesario activar los null que el objeto recibido pueda tener, además también los recupera para los objetos referenciados hasta una profundidad máxima por defecto.

Los objetos de la clase `SimpleConstraint` permiten establecer criterios de búsqueda simples que serán utilizados dentro de los objetos `Query`. La clase `SimpleConstraint` no ofrece constructor, pero sí ofrece unos métodos de clase que se encargan de construir distintos tipos de restricciones que hay:

- SimpleConstraint newEqualConstraint(String field, Object value)
- SimpleConstraint newNotEqualConstraint(String field, Object value)
- SimpleConstraint neGreaterThanConstraint(String field, Object value)
- SimpleConstraint newGreaterThanOrEqualsConstraint(String field, Object value)
- SimpleConstraint newLessThanConstraint(String field, Object value)
- SimpleConstraint newLessThanOrEqualsConstraint(String field, Object value)

Teniendo estas condiciones simples es posible, si se desea, crear condiciones más complejas usando los distintos operadores. Estos operadores tienen el constructor sobrecargado, pudiendo ser llamado con un número variable o una lista de `Constraint`, excepto la restricción `NotConstraint` que solo recibe una condición y la devuelve negada:

- `AndConstraint(Constraint... c)`
- `AndConstraint(List<Constraint> l)`
- `OrConstraint(Constraint... c)`
- `OrConstraint(List<Constraint> l)`
- `NotConstraint(Constraint c)`

Apéndice 2: Contribuciones personales

Victor Delgado Fernandez

- Investigue tecnologías parecidas que habían resuelto nuestro problema y para tomar ideas.
- Buscar las herramientas más adecuadas o familiares para poder llevar a cabo nuestro proyecto.
- Discutir las ventajas e inconvenientes de cada una de las tecnologías buscadas para ver cual se adecuaba más a nuestro proyecto.
- Instalación de los entornos y herramientas elegidas para poder trabajar en nuestro proyecto.
- Estudio de los entornos y herramientas elegidas porque con alguna de ellas no estábamos familiarizados.
- Primeros pasos con la librería incluyendo su constructor y métodos que usa para conectarse a la base de datos.
- Construcción de una tabla de índice para saber qué atributos y con qué nombre se guardan en cada tabla.
- Creación de un método llamado `save` para insertar los objetos simples en la base de datos.
- Realización del método `delete` para borrar el objeto que le pases en su tabla de la base de datos.
- Cambiar el código de `save` para que ahora guarde el objeto que le pases con referencias simples.

- Modificar el método `save` para que guarde la clase real con el que se crearon los objetos(problema clases abstractas)

- Refactorizacion de codigo para que las excepciones lleguen al usuario, además crearnos excepciones propias para cuando vuelves a guardar un objeto que ya está guardado y otra para cuando vayas a borrar un objeto que no se haya guardado ni recuperado.
- Añadir la funcionalidad de profundidad cuando vayamos a recuperar un objeto que esté en la base de datos.
- Realización del método de vaciar una tabla intermedia cuando se vaya a guardar un objeto ya guardado anteriormente.
- Añadir la funcionalidad necesaria para guardar `List<>` y `Array` tando de objetos simples (multivalorado), como objetos no simples (relaciones de multiplicidad).

Carlos Fernández Bravo

- Investigación de patrones de diseño que se aplicarán.
- Construcción de la interfaz `Constraint` que se utilizará para implementar los tipos de restricciones.
- Implementación de las restricciones simples denominadas `SimpleConstraint` que incluyen las restricciones de igual, mayor, menor, mayor o igual, menor o igual y distinto.
- Implementación de las restricciones de tipo `AND` y `OR` que hacen la intersección y la unión respectivamente de cualquier tipo de restricción.
- Implementación de la restricción de tipo `NOT` que hace la negación de la restricción a la que afecta.
- Diseño e implementación siguiendo el patrón `Query Object` de la clase `Query`.
- Diseño e implementación del método `toSql` que devuelve la sentencia SQL a ejecutar a partir de las restricciones aplicadas.
- Diseño e implementación del método `executeQuery` que devuelve una lista de objetos recibidos y construidos al ejecutar la sentencia SQL del método nombrado anteriormente.
- Diseño e implementación del constructor de objetos para reconstruir un objeto con atributos simples a partir de la información recuperada de la base de datos.
- Creación de los mapas de identidad a la clase principal de la librería.
- Creación de un mapa en la clase principal de la librería con el nombre de la clase y su correspondiente tabla para reducir el número de conexiones a la base de datos para hacer consulta sobre estos datos.
- Creación del método `basicType` para conocer si un atributo es de tipo básico desde cualquier clase incluida en la librería.

- Incorporación de los mapas de identidad en el método de recuperación para solucionar el problema de la identidad de objetos.
- Diseño e implementación del método `queryByExample`.
- Extensión del método `queryByExample` para permitir usar restricciones de cero y nulo sobre los atributos de un objeto.
- Extensión de los métodos de la clase `Query` para permitir hacer restricciones de identidad sobre un atributo de tipo no básico de un objeto.
- Extensión de los métodos de la clase `Query` para permitir hacer restricciones sobre los atributos pertenecientes a atributos de tipo no básico de un objeto sin límite de nivel de profundidad.
- Extensión del método constructor de objetos para incluir la construcción de los atributos de tipo no básico.
- Extensión de los métodos de la clase `Query` para recuperar atributos de tipo `List` pertenecientes a un objeto manteniendo el orden en el que se encontraban en la lista que fue guardada.
- Extensión de los métodos de la clase `Query` para recuperar atributos de tipo array pertenecientes a un objeto manteniendo el orden en el que se encontraban en el array que fue guardado.
- Extensión de los métodos de la clase `Query` para recuperar atributos de tipo `Set`.
- Extensión del método constructor de objetos para incluir la construcción de los atributos de tipo `List`, `Set` y array.
- Extensión de los métodos de la clase `Query` para permitir hacer consultas sobre si un atributo de tipo `List`, `Set` o array contiene un objeto.

Álvaro Isabel Torija

- Investigación de tecnologías parecidas que habían resuelto nuestro problema de guardar objetos en bases de datos para tomar ideas.
- Buscar las herramientas más adecuadas o familiares para poder llevar a cabo nuestro proyecto.
- Discutir las ventajas e inconvenientes de cada una de las tecnologías buscadas para ver cual se adecuaba más a nuestro proyecto.
- Instalación de los entornos y herramientas elegidas para poder trabajar en nuestro proyecto.
- Estudio de los entornos y herramientas elegidas porque con alguna de ellas no estábamos familiarizados.
- Construcción de una tabla de índice de tablas y un método que dado un objeto, compruebe para el nombre de su clase si ya está en el índice de tablas, y si no está lo inserte junto con el nombre que tendrá su tabla, en cualquier caso devuelve el nombre de la tabla que se usa para guardar objetos de esa clase.
- Añadir al método guardar un método que altera la tabla donde se guardan los objetos de esa clase, para que coincida el número de columnas y su tipo con el número de atributos y su tipo del objeto a guardar.
- Añadir el mapa `indentytyMap` de objetos al método guardar para que no te deje guardar objetos ya guardados ni idénticos
- Realización del método `update` para actualizar los objetos simples ya insertados o cargados en la base de datos
- Resolver el problema de los ciclos en las referencias entre objetos, usando mapa de objetos visitados, para no volver a guardar objetos ya guardados.
- Modificar el método `save` para obtener la id del objeto recién insertado usando `getGeneratedKeys()` en vez de ejecutar otra consulta.

- Refactorizar el código para añadir una excepción `RunTimeExeption` llamada `OOBDLibraryException` envoltorio de todas las demás excepciones, esta será la única excepción que lanzará la librería.
- Crear método `actívate` con profundidad, para recuperar las referencias nulas que pudieran tener los objetos recuperados con una profundidad máxima.
- Añadir la funcionalidad necesaria para guardar `Set<>` tando de objetos simples, guardados como si fueran atributos multivalorados, como objetos no simples, guardados como si fueran relaciones de multiplicidad.

Bibliografía

- [1]: Patrick Peak, Nick Heudecker, Hibernate in Action: Practical Object/Relational Mapping, Paperback – August 1, 2004
- [2]: Artículo de bd4o en la Wikipedia, https://en.wikipedia.org/wiki/Db4o
- [3]: Página oficial de ormlite, http://ormlite.com/
- [4]: Artículo de OJB en la Wikipedia, https://es.wikipedia.org/wiki/OJB
- [5]: Presentación de introducción a ObjectStore, http://www.odbms.org/wp-content/uploads/2013/11/ObjectStoreIntro-v2.pdf
- [6]: Ira R. Forman and Nate Forman, Java Reflection in Action, Manning - 2004
- [7]:George Reese, Java Database Connectivity, O'Reilly, 1997
- [8]:Martin Fowler, Patterns of Enterprise Application Architecture, Addison Wesley, 2003